Hibernate Search 6.0.11.Final

Migration Guide from 5.11

2023-01-31

Table of Contents

Introduction ... 1
Recommended procedure ... 2
Requirements ... 3
Maven coordinates changes ... 4
Data format and schema changes ... 5
Migration helper ... 6
Purpose ... 6
How to use the migration helper ... 6
Limitations of the migration helper ... 7
Configuration changes ... 8
Basics ... 8
Constants for property keys ... 8
Configuration property reference ... 9
JMX ... 18
Backends ... 18
API changes ... 20
Annotation mapping ... 20
`@Analyzer` ... 20
`@AnalyzerDef`, `@AnalyzerDefs` ... 20
`@AnalyzerDiscriminator` ... 20
`@Boost` ... 20
`@CacheFromIndex` ... 21
`@CalendarBridge` ... 21
`@CharFilterDef` ... 21
`@ClassBridge`, `@ClassBridges` ... 21
`@ContainedIn` ... 21
`@DateBridge` ... 22
`@DocumentId` ... 22
`@DynamicBoost` ... 23
`@Facet`, `@Facets` ... 23
Basics ... 23
`@Facet.encoding` ... 24
`@Field`, `@Fields` ... 24
Basics ... 24
`@Field.analyze` ... 25
`@Field.analyzer` ... 25
`@Field.boost` ... 25
`@Field.bridge` ... 26
`@Field.index` ... 26
`@Field.indexNullAs` ... 26

@Field.name ... 27
@Field.normalizer ... 27
@Field.norms ... 28
@Field.store ... 28
@Field.termVector ... 28
@FieldBridge ... 29
@FullTextFilterDef, @FullTextFilterDefs ... 29
@Indexed ... 29
Basics ... 29
@Indexed.index ... 29
@Indexed.interceptor ... 29
@IndexedEmbedded ... 29
Basics ... 29
@IndexedEmbedded.depth ... 29
@IndexedEmbedded.includePaths ... 30
@IndexedEmbedded.prefix ... 30
@IndexedEmbedded.targetElement ... 30
@IndexedEmbedded.indexNullAs ... 31
@IndexedEmbedded.includeEmbeddedObjectId ... 31
Using @IndexedEmbedded to request container extraction ... 31
@Key ... 32
@Latitude ... 32
@Longitude ... 32
@Normalizer ... 32
@NormalizerDef, @NormalizerDefs ... 32
@NumericField, @NumericFields ... 32
@ProvidedId ... 33
@SortableField, @SortableFields ... 33
@Spatial, @Spatials ... 33
Basics ... 33
@Spatial.boost ... 35
@Spatial.name ... 35
@Spatial.spatialMode, @Spatial.topSpatialHashLevel, @Spatial.bottomSpatialHashLevel ... 36
@Spatial.store ... 36
@TikaBridge ... 37
@TokenFilterDef ... 37
@TokenizerDef ... 37
Programmatic mapping ... 37
Analysis definition provider ... 37
Bridges ... 38
Entity indexing interceptors ... 38

FullTextEntityManager/FullTextSession ... 39
Basics ... 39
FullTextSession.createFullTextQuery ... 39
FullTextSession.index(…), FullTextSession.purge(…) ... 39
FullTextSession.flushToIndexes() ... 40
FullTextSession.purgeAll(…) ... 40
FullTextSession.createIndexer(…) ... 41
Searching ... 41
FullTextQuery ⇒ DSL ... 41
Basics ... 41
Adapter from search query to JPA/ORM query ... 42
FullTextQuery.initializeObjectsWith ... 42
FullTextQuery.setCriteriaQuery(…) ... 42
FullTextQuery.explain(…) ... 43
FullTextQuery.setFilter(…) ... 43
FullTextQuery.enableFullTextFilter(…) / FullTextQuery.disableFullTextFilter(…) 43
org.apache.lucene.search.Query ⇒ SearchPredicate ... 43
Basics ... 43
Query DSL migration reference ... 45
Native query ... 48
org.apache.lucene.search.Sort / SortField ⇒ SearchSort ... 48
Sort DSL migration reference ... 49
Native sorts ... 51
Projections ... 51
Basics ... 51
ProjectionConstants/ElasticsearchProjectionConstants migration reference ... 52
Facets ⇒ Aggregations ... 54
Discrete faceting ... 55
Range faceting ... 56
Drill-down with .selectFacets ... 57
Error handler ... 57
Full-text filter ... 57
@Factory ... 59
SearchException ... 60
Sharding: IndexShardingStrategy/ShardIdentifierProvider /ShardIdentifierProviderTemplate 60
SearchFactory ... 60
Basics ... 60
Merging segments: SearchFactory.optimize()/SearchFactory.optimize(…) ... 61
Getting Lucene analyzers: SearchFactory.getAnalyzer(…) ... 61

Building queries: `SearchFactory.buildQueryBuilder()` 62
Statistics: `SearchFactory.getStatistics()` 62
Accessing Lucene index readers: `SearchFactory.getIndexReaderAccessor` 62
Metamodel: `SearchFactory.getIndexedTypeDescriptor`, `SearchFactory.getIndexedTypes` 62
Accessing the Elasticsearch client: `SearchFactory.getIndexFamily` 62
`MassIndexer` 63
Basics 63
`MassIndexer.optimizeOnFinish(…)` 63
`MassIndexer.optimizeAfterPurge(…)` 63
`MassIndexer.progressMonitor(…)` 63
`MassIndexer.threadsForSubsequentFetching(…)` 63
Batch (JSR-352) integration 64
SPI changes 65
Behavior changes 66
No default bridge for `java.util.Class` 66
The document ID is not an index field 66
`@Indexed` is inherited 66
One index ⇒ one type 67
The index name defaults to the entity name 67
`indexNullAs` is irrelevant when building predicates and when projecting 68
Faceting returns normalized strings 68
`@IndexedEmbedded` on an association enables reindexing when the target entity changes by default 69
Indexed, `@Transient` properties require additional configuration 70
Container extraction for `@*Field` is implied 70
More fields are numeric by default 71
Scrolling is forward-only 71
Scrolling/`Query.iterate()` load entities eagerly in batches 72
When sorting by field value or distance, missing values are last by default 72
Logger `org.hibernate.search.query` uses the `TRACE` level 72
The default similarity for Lucene is now BM25 72

Introduction

The aim of this guide is to assist you migrating an existing application using any version `5.11.x` of Hibernate Search to the latest of the `6.0.x` series.

If you think something is missing or something does not work, please contact us.

If you're looking to migrate from an earlier version, you should migrate step-by-step, from one minor version to the next, following the migration guide of each version.

Recommended procedure

Search 6 introduces new APIs, so migrating older projects will be more work than usual.

To facilitate the process of migrating, Hibernate Search 6 includes a "migration helper" module that allows you to use the Hibernate Search 5 APIs with Hibernate Search 6 and a Lucene backend under the hood. However, this module does not offer full backward compatibility: for some features that changed dramatically, it may not be possible to use the Search 5 APIs anymore. See Migration helper for more information.

For those who cannot afford to, or do not want to, spend the time required to migrate, we intend to continue maintenance releases (= bugfixes) of Hibernate Search 5.x: no end-of-life date has been set at the moment.

Requirements

Hibernate Search 6 is still compatible with both JDK8 and JDK11.

The required versions of dependencies changed:

- The Hibernate ORM mapper now requires Hibernate ORM 5.4.32.Final (5.4.3.Final and earlier won't work correctly).
- The Elasticsearch backend now requires Elasticsearch 5.6, 6.8 or 7.10.
- The Lucene backend now requires Lucene 8.7.0.

Maven coordinates changes

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

- the group IDs changed from `org.hibernate` to `org.hibernate.search`
- most of the artifact IDs changed to reflect the new mapper/backend design
- the Lucene integration now requires an explicit dependency instead of being pulled by the engine by default.

Read the getting started guide, section "dependencies" for more information.

Data format and schema changes

Indexes created with Hibernate Search 5 or earlier are not compatible with Hibernate Search 6. This goes for embedded-Lucene indexes as well as Elasticsearch indexes.

In order to upgrade an application to Hibernate Search 6, all data must be reindexed. See the documentation of the `MassIndexer` for instructions.

Similarly, native queries/predicates/sorts targeting Hibernate Search 5 indexes (e.g. manual instantiation of Lucene `Query` types, Elasticsearch JSON) may not work correctly with Hibernate Search 6, because the underlying type of some fields may have changed. To avoid that sort of problem in future major upgrades, we recommend you use the Hibernate Search DSL to create predicates and sorts, so that Hibernate Search will automatically pick the right predicate/sort depending on the field type.

Migration helper

Purpose

Hibernate Search 6 includes a temporary additional "migration helper" module that provides partial compatibility with Hibernate Search 5 APIs backed by the Hibernate Search 6 implementations.

This module should make migration easier by making sure that code relying on the most-frequently-used APIs (mapping annotations, search DSL, …) continues to compile and run. The idea is to use the migration helper temporarily to make most of the application code (search queries, …) work, making it easier to focus on migrating configuration and to assess the effort required to migrate the remaining code.

The migration helper only works with a Lucene backend.

Hibernate Search 5 APIs are not well suited to Elasticsearch, so applications relying on Hibernate Search 5's experimental Elasticsearch integration should migrate to the more adapted Hibernate Search 6 APIs right away.

The migration helper should not be used in production environments.

It has limitations preventing full compatibility with Hibernate Search 5, and these limitations will never be addressed.

All APIs defined in the migration helper are deprecated and will be removed in the next major version of Hibernate Search.

How to use the migration helper

To use the migration helper, add the following dependency to your project:

```
<dependency>
   <groupId>org.hibernate.search</groupId>
   <artifactId>hibernate-search-v5migrationhelper-orm</artifactId>
   <version>6.0.11.Final</version>
</dependency>
```

Then, try to recompile your application. Compilation errors should point you to the most significant API changes that require your immediate attention; most of the code that still compiles should work as it used to in Hibernate Search 5.

Limitations of the migration helper

First, the migration helper **only works with a Lucene backend**. Hibernate Search 5 APIs are not well suited to Elasticsearch, so applications relying on Hibernate Search 5's experimental Elasticsearch integration should migrate to the more adapted Hibernate Search 6 APIs right away.

Second, the migration helper only addresses Java API compatibility. This excludes in particular:

- Configuration properties: they must still be replaced with Search 6 properties.
- Data format: data must still be reindexed.
- Behavior: most of the behavior changes in Hibernate Search 6 also affect the migration helper. In particular, One index ⇒ one type, `@Indexed` is inherited and The document ID is not an index field.

Finally, even for Java APIs, not **all** APIs from Hibernate Search 5 are present in the migration helper. You should get compilation errors when APIs are missing, which should make them easy to spot, but here are the main pitfalls:

- APIs that have no equivalent in Hibernate Search 6 are not available in the migration helper.

  This includes in particular full-text filters, index-time boost, statistics, `IndexReaderAccessor`, referring to analyzers by class, annotation-based analyzer definitions (`@AnalyzerDef`, `@AnalyzerDefs`/`@NormalizerDef`, `@NormalizerDefs`), `@AnalyzerDiscriminator`, …

- Hibernate Search 5 bridges cannot be used in the migration helper. You can however mix Hibernate Search 5 annotations with Hibernate Search 6 annotations, so feel free to migrate only annotations where a custom bridge is used (see `@Field`, `@Fields`), as a first step.
- Hibernate Search 5 programmatic mapping cannot be used in the migration helper. You should move to the Hibernate Search 6 programmatic mapping, as explained here.
- Hibernate Search 5 metamodel APIs cannot be used in the migration helper. You should move to the Hibernate Search 6 metamodel APIs, as explained in Metamodel: `SearchFactory.getIndexedTypeDescriptor`, `SearchFactory.getIndexedTypes`.

Configuration changes

Basics

Most configuration properties changed in Hibernate Search 6. In most cases it's only a matter of changing the prefix of a configuration property (due to the different structure of Hibernate Search 6), but in a few cases the relevant feature changed so much that a new approach was necessary for configuration.

For a quick introduction to the basics of configuration in Hibernate Search 6, refer to the getting started guide, section "configuration".

For more details, see the main "configuration" section of the reference documentation.

For a complete list of Hibernate Search 5 properties and their equivalent in Hibernate Search 6, refer to the section below.

Constants for property keys

In Hibernate Search 5, constants for configuration property keys used to be provided through `org.hibernate.search.cfg.Environment` and `org.hibernate.search.elasticsearch.cfg.ElasticsearchEnvironment`.

In Hibernate Search 6, constants are provided through classes whose name ends with `Settings`:

- `org.hibernate.search.engine.cfg.EngineSettings`
- `org.hibernate.search.engine.cfg.BackendSettings`
- `org.hibernate.search.engine.cfg.IndexSettings`
- `org.hibernate.search.mapper.orm.cfg.HibernateOrmMapperSettings`
- `org.hibernate.search.backend.lucene.cfg.LuceneBackendSettings`
- `org.hibernate.search.backend.lucene.cfg.LuceneIndexSettings`
- `org.hibernate.search.backend.elasticsearch.cfg.ElasticsearchBackendSettings`
- `org.hibernate.search.backend.elasticsearch.cfg.ElasticsearchIndexSettings`
- `org.hibernate.search.backend.elasticsearch.aws.cfg.ElasticsearchAwsBackendSettings`

Configuration property reference

Below is a list of Hibernate Search 5 properties in alphabetical order, along with their equivalent in Hibernate Search 6.

Index defaults are no longer specified using the prefix `hibernate.search.default.`, and `hibernate.search.indexes.default.` will not work either.

To specify configuration to be applied by default to all indexes, just set the configuration at the backend level using the prefix `hibernate.search.backend.`.

`hibernate.search.analyzer`

No direct equivalent in Hibernate Search 6.

To override the default analyzer, define a custom analyzer named `default`. See Analysis definition provider.

`hibernate.search.autoregister_listeners`

Hibernate Search 6 equivalent: `hibernate.search.enabled`.

`hibernate.search.batch_size`

No direct equivalent in Hibernate Search 6.

This property was not documented in Hibernate Search 5. For the specific use case of batch processes, know that upon Hibernate ORM session flushes, Hibernate Search 6 will automatically turn entities to documents and hold documents in memory until the transaction commit.

See also this section of the documentation.

`hibernate.search.default.elasticsearch.aws.*`

The syntax for configuring AWS authentication changed slightly in Hibernate Search 6.

In short, the following configuration in Hibernate Search 5:

```
hibernate.search.default.elasticsearch.aws.signing.enabled = true
hibernate.search.default.elasticsearch.aws.region = us-east-1
hibernate.search.default.elasticsearch.aws.access_key = AKIDEXAMPLE
hibernate.search.default.elasticsearch.aws.secret_key =
wJalrXUtnFEMI/K7MDENG+bPxRfiCYEXAMPLEKEY
```

Will look like this in Hibernate Search 6:

```
hibernate.search.backend.aws.signing.enabled = true
hibernate.search.backend.aws.region = us-east-1
hibernate.search.backend.aws.credentials.type = static
hibernate.search.backend.aws.credentials.access_key_id = AKIDEXAMPLE
hibernate.search.backend.aws.credentials.secret_access_key =
wJalrXUtnFEMI/K7MDENG+bPxRfiCYEXAMPLEKEY
```

Other types of credentials are available in Hibernate Search 6, see this section of the documentation

`hibernate.search.default.elasticsearch.connection_timeout`

Hibernate Search 6 equivalent: `hibernate.search.backend.connection_timeout`.

Defaults to 1000 (1 second) in Hibernate Search 6 instead of 3 seconds in Hibernate Search 5. More information here.

`hibernate.search.default.elasticsearch.discovery.default_scheme`

Hibernate Search 6 equivalent: `hibernate.search.backend.protocol`.

`hibernate.search.default.elasticsearch.discovery.enabled`

Hibernate Search 6 equivalent: `hibernate.search.backend.discovery.enabled`.

`hibernate.search.default.elasticsearch.discovery.refresh_interval`

Hibernate Search 6 equivalent: `hibernate.search.backend.discovery.refresh_interval`.

`hibernate.search.default.elasticsearch.dynamic_mapping`, `hibernate.search.<index-name>.elasticsearch.dynamic_mapping`

Hibernate Search 6 equivalent (global defaults): `hibernate.search.backend.dynamic_mapping`.

Hibernate Search 6 equivalent (per-index): `hibernate.search.backend.indexes.<index-name>.dynamic_mapping`.

`hibernate.search.default.elasticsearch.host`

Hibernate Search 6 equivalent: `hibernate.search.backend.uris`.

In Hibernate Search 6, the property `hibernate.search.backend.hosts` can also be used, but the URI scheme (`http://` or `https://`) must **not** be included there; either use `hibernate.search.backend.uris` and include the URI scheme directly in URIs, or use `hibernate.search.backend.hosts` without the URI scheme and specify the protocol through the separate property `hibernate.search.backend.protocol`. See this section of the documentation for more details.

`hibernate.search.default.elasticsearch.index_management_wait_timeout`, `hibernate.search.<index-name>.elasticsearch.index_management_wait_timeout`

Hibernate Search 6 equivalent (global defaults): `hibernate.search.backend.schema_management.minimal_required_status_wait_timeout`.

Hibernate Search 6 equivalent (per-index): `hibernate.search.backend.indexes.<index-name>.schema_management.minimal_required_status_wait_timeout`.

`hibernate.search.default.elasticsearch.index_schema_management_strategy`, `hibernate.search.<index-name>.elasticsearch.index_schema_management_strategy`

Hibernate Search 6 equivalent (global defaults): `hibernate.search.schema_management.strategy`.

Hibernate Search 6 equivalent (per-index): none.

Defaults to `create-or-validate` in Hibernate Search 6. See schema management.

There is no direct equivalent for the per-index variant in Hibernate Search 6: automatic schema management is configured globally for all indexes, not on a per-index basis. However, you can achieve more control by setting `hibernate.search.schema_management.strategy` to `none` and managing the schema manually after startup.

`hibernate.search.default.elasticsearch.max_total_connection_per_route`

Hibernate Search 6 equivalent: `hibernate.search.backend.max_connections_per_route`.

`hibernate.search.default.elasticsearch.max_total_connection`

Hibernate Search 6 equivalent: `hibernate.search.backend.max_connections`.

`hibernate.search.default.elasticsearch.password`

Hibernate Search 6 equivalent: `hibernate.search.backend.password`.

`hibernate.search.default.elasticsearch.path_prefix`

Hibernate Search 6 equivalent: `hibernate.search.backend.path_prefix`.

`hibernate.search.default.elasticsearch.read_timeout`

Hibernate Search 6 equivalent: `hibernate.search.backend.read_timeout`.

Defaults to 30000 (30 seconds) in Hibernate Search 6 instead of 60 seconds in Hibernate Search 5. More information here.

`hibernate.search.default.elasticsearch.refresh_after_write`, `hibernate.search.<index-name>.elasticsearch.refresh_after_write`

Hibernate Search 6 equivalent (global defaults):

`hibernate.search.automatic_indexing.synchronization.strategy`.

Hibernate Search 6 equivalent (per-index): none.

Setting `hibernate.search.automatic_indexing.synchronization.strategy` to `read-sync` or `sync` will produce results similar to setting `hibernate.search.default.elasticsearch.refresh_after_write` to `true`. See automatic indexing synchronization for more information.

There is no equivalent for the per-index variant in Hibernate Search 6: the synchronization strategy can only be set globally, not on a per-index basis.

`hibernate.search.default.elasticsearch.request_timeout`

Hibernate Search 6 equivalent: `hibernate.search.backend.request_timeout`.

Defaults to no timeout in Hibernate Search 6. More information here.

`hibernate.search.default.elasticsearch.required_index_status`, `hibernate.search.<index-name>.elasticsearch.required_index_status`

Hibernate Search 6 equivalent (global defaults): `hibernate.search.backend.schema_management.minimal_required_status`.

Hibernate Search 6 equivalent (per-index): `hibernate.search.backend.indexes.<index-name>.schema_management.minimal_required_status`.

`hibernate.search.default.elasticsearch.username`

Hibernate Search 6 equivalent: `hibernate.search.backend.username`.

`hibernate.search.default.exclusive_index_use`, `hibernate.search.<index-name>.exclusive_index_use`

No equivalent in Hibernate Search 6.

`hibernate.search.default.directory_provider`, `hibernate.search.<index-name>.directory_provider`

Hibernate Search 6 equivalent (global defaults): `hibernate.search.backend.directory.type`.

Hibernate Search 6 equivalent (per-index): `hibernate.search.backend.indexes.<index-name>.directory.type`.

Values for this property may have changed:

- `filesystem` ⇒ `local-filesystem`
- `local-heap` ⇒ `local-heap`

- `ram` ⇒ `local-heap`
- `filesystem-master`, `filesystem-slave`, `infinispan` ⇒ no longer supported

`hibernate.search.default.indexBase`, `hibernate.search.<index-name>.indexBase`

Hibernate Search 6 equivalent (global defaults): `hibernate.search.backend.directory.root`.

Hibernate Search 6 equivalent (per-index): `hibernate.search.backend.indexes.<index-name>.directory.root`.

`hibernate.search.default.indexName`, `hibernate.search.<index-name>.indexName`

No equivalent in Hibernate Search 6.

The name of an index can still be customized in the mapping, using `@Indexed(name = …)`, or with the programmatic equivalent.

`hibernate.search.default.index_flush_interval`, `hibernate.search.<index-name>.index_flush_interval`

Hibernate Search 6 equivalent (global defaults): `hibernate.search.backend.io.commit_interval`.

Hibernate Search 6 equivalent (per-index): `hibernate.search.backend.indexes.<index name>.io.commit_interval`.

`hibernate.search.default.index_metadata_complete`, `hibernate.search.<index-name>.index_metadata_complete`

No equivalent in Hibernate Search 6.

This property was not documented in Hibernate Search 5.

`hibernate.search.default.indexmanager`, `hibernate.search.<index-name>.indexmanager`

Hibernate Search 6 equivalent (global defaults): `hibernate.search.backend.type`.

Hibernate Search 6 equivalent (per-index): none.

Setting the backend type (`elasticsearch` or `lucene`) should no longer be necessary: it will be picked automatically if there is only one backend type available in the classpath.

If you have multiple backend types available in the classpath for some reason, but only want to use one, set `hibernate.search.backend.type` to either `lucene` or `elasticsearch`.

If you need both a Lucene backend and an Elasticsearch backend, proceed as follows:

- Annotate entities that must be indexed in the Elasticsearch backend with `@Indexed(backend = "elasticsearch")`.
- Annotate entities that must be indexed in the Lucene backend with `@Indexed(backend = "lucene")`.
- Configure two separate backends in your configuration properties:
  - prefix properties of the Elasticsearch backend with `hibernate.search.backends.elasticsearch.` instead of `hibernate.search.backend.`.
  - prefix properties of the Lucene backend with `hibernate.search.backends.lucene.` instead of `hibernate.search.backend.`.
  - same goes for indexes, e.g. `hibernate.search.backends.elasticsearch.indexes.<index-name>.someProperty` for indexes of the Elasticsearch backend or `hibernate.search.backends.lucene.indexes.<index-name>.someProperty` for indexes of the Lucene backend.

`hibernate.search.default.indexwriter.`, `hibernate.search.<index-name>.indexwriter.`

Hibernate Search 6 equivalent (global defaults): `hibernate.search.backend.io.writer.` or `hibernate.search.backend.io.merge.`.

Hibernate Search 6 equivalent (per-index): `hibernate.search.backend.indexes.<index-name>.io.writer.` or `hibernate.search.backend.indexes.<index-name>.io.merge.`.

The writer settings and merge settings are now split. See here for available writer settings and here for available merge settings.

`hibernate.search.default.locking_strategy`, `hibernate.search.<index-name>.locking_strategy`

Hibernate Search 6 equivalent (global defaults): `hibernate.search.backend.directory.locking.strategy`.

Hibernate Search 6 equivalent (per-index): `hibernate.search.backend.indexes.<index-name>.directory.locking.strategy`.

See here for available locking strategies.

`hibernate.search.default.max_queue_length`, `hibernate.search.<index-name>.max_queue_length`

Hibernate Search 6 equivalent (global defaults): `hibernate.search.backend.indexing.queue_size`.

Hibernate Search 6 equivalent (per-index): `hibernate.search.backend.indexes.<index-name>.indexing.queue_size`.

In Hibernate Search 6, there are multiple queues per index, enabling parallel indexing of documents. See here for Lucene or here for Elasticsearch.

`hibernate.search.default_null_token`

No equivalent in Hibernate Search 6.

In most cases, you won't need to use `indexNullAs` anymore. Where `indexNullAs` is still needed, define the token explicitly for each index field.

`hibernate.search.default.reader.`, `hibernate.search.<index-name>.reader.`

No direct equivalent in Hibernate Search 6.

To enable async reader refresh, set `hibernate.search.backend.io.refresh_interval` or `hibernate.search.backend.indexes.<index-name>.io.refresh_interval` to a strictly positive value (in milliseconds). See here for more information.

Custom reader strategies are no longer supported.

`hibernate.search.default.retry_marker_lookup`, `hibernate.search.<index-name>.retry_marker_lookup`

No equivalent in Hibernate Search 6: the `filesystem-slave` directory provider is no longer supported.

`hibernate.search.default.sharding_strategy`, `hibernate.search.<index-name>.sharding_strategy`

No direct equivalent in Hibernate Search 6: sharding is configured differently. See Sharding: `IndexShardingStrategy`/`ShardIdentifierProvider`/`ShardIdentifierProviderTemplate`.

`hibernate.search.default.sharding_strategy.nbr_of_shards`, `hibernate.search.<index-name>.sharding_strategy.nbr_of_shards`

No direct equivalent in Hibernate Search 6: sharding is configured differently. See Sharding: `IndexShardingStrategy`/`ShardIdentifierProvider`/`ShardIdentifierProviderTemplate`.

`hibernate.search.default.similarity`, `hibernate.search.<index-name>.similarity`

No direct equivalent in Hibernate Search 6: the similarity is configured through the analysis configurer.

`hibernate.search.default.worker.backend`, `hibernate.search.<index-name>.worker.backend`

No equivalent in Hibernate Search 6: the JMS/JGroups backends are no longer supported.

`hibernate.search.default.worker.execution`, `hibernate.search.<index-name>.worker.execution`

No direct equivalent in Hibernate Search 6.

Setting `hibernate.search.automatic_indexing.synchronization.strategy` to `async` or `sync` will produce results similar to setting `hibernate.search.<index-name>.worker.execution` to the same value. See automatic indexing synchronization for more information.

`hibernate.search.default.worker.*, hibernate.search.<index-name>.worker.*`

No equivalent in Hibernate Search 6: the JMS/JGroups backends are no longer supported.

`hibernate.search.elasticsearch.analysis_definition_provider`

Hibernate Search 6 equivalent: `hibernate.search.backend.analysis.configurer`.

A different interface should be implemented: see Analysis definition provider.

`hibernate.search.elasticsearch.log.json_pretty_printing`

Hibernate Search 6 equivalent: `hibernate.search.backend.log.json_pretty_printing`.

`hibernate.search.elasticsearch.scroll_backtracking_window_size`

No equivalent in Hibernate Search 6: scrolling is forward-only.

`hibernate.search.elasticsearch.scroll_fetch_size`

No direct equivalent in Hibernate Search 6.

When using Hibernate Search APIs, the "chunk size" is an argument to the `scroll` method. When using the Hibernate ORM or JPA adapters, the "chunk size" is set to the same value as the fetch size.

`hibernate.search.elasticsearch.scroll_timeout`

Hibernate Search 6 equivalent: `hibernate.search.backend.scroll_timeout`.

`hibernate.search.enable_dirty_check`

Hibernate Search 6 equivalent: `hibernate.search.automatic_indexing.enable_dirty_check`.

`hibernate.search.error_handler`

Hibernate Search 6 equivalent: `hibernate.search.background_failure_handler`.

A different interface should be implemented: see Error handler.

`hibernate.search.filter.cache_docidresults.size`

No equivalent in Hibernate Search 6.0. See Full-text filter.

If you need caching for some of your Lucene queries, consider upgrading directly to Hibernate Search 6.1, which provides configurable Low-level hit caching.

`hibernate.search.filter.cache_strategy`

No equivalent in Hibernate Search 6.0. See Full-text filter.

If you need caching for some of your Lucene queries, consider upgrading directly to Hibernate Search 6.1, which provides configurable Low-level hit caching.

`hibernate.search.generate_statistics`

No equivalent in Hibernate Search 6. See Statistics: `SearchFactory.getStatistics()`.

`hibernate.search.index_uninverting_allowed`

Index uninverting was deprecated in Hibernate Search 5 due to poor performance and is no longer allowed. All index fields that you want to sort on must be marked as sortable.

`hibernate.search.indexing_strategy`

Hibernate Search 6 equivalent: `hibernate.search.automatic_indexing.strategy`.

Set to `none` to get the equivalent of `hibernate.search.indexing_strategy = manual` in Hibernate Search 5.

`hibernate.search.jmx_bean_suffix`

No equivalent in Hibernate Search 6. See JMX.

`hibernate.search.jmx_enabled`

No equivalent in Hibernate Search 6. See JMX.

`hibernate.search.lucene.analysis_definition_provider`

Hibernate Search 6 equivalent: `hibernate.search.backend.analysis.configurer`.

A different interface should be implemented: see Analysis definition provider.

`hibernate.search.lucene_version`

Hibernate Search 6 equivalent: `hibernate.search.backend.lucene_version`.

`hibernate.search.model_mapping`

Hibernate Search 6 equivalent: `hibernate.search.mapping.configurer`.

A different interface should be implemented: see Programmatic mapping.

`hibernate.search.query.database_retrieval_method`

No equivalent in Hibernate Search 6: entities are always loaded with a query.

`hibernate.search.query.object_lookup_method`

Hibernate Search 6 equivalent: `hibernate.search.query.loading.cache_lookup.strategy`.

See this section of the documentation.

`hibernate.search.similarity`

No direct equivalent in Hibernate Search 6: the similarity is configured through the analysis configurer.

The default similarity when this property is not set changed; see The default similarity for Lucene is now BM25.

`hibernate.search.worker.*`

No equivalent to the concept of "worker" in Hibernate Search 6:

- automatic indexing is always performed on transaction commit or, when there is no transaction, on session flush.
- transactional backends, for example the JMS backend, are no longer supported.

JMX

Hibernate Search 6 does not provide JMX support at the moment.

The current plans are to implement support for tracing in a future release (HSEARCH-4057). This would provide a more powerful solution to users looking for insight into the behavior of their application.

If you need this feature urgently, we'll gladly help anyone interested in contributing a patch: feel free to contact us.

Backends

Hibernate Search 6 does not provide support for the JGroups or JMS backends at the moment, nor does it support the related `filesystem-master`/`filesystem-slave`/`infinispan` directory providers.

If you need to scale your application to multiple nodes, consider switching to the Elasticsearch backend.

To ensure robustness of distributed indexing, consider upgrading directly to Hibernate Search 6.1, which provides a feature called coordination, and in particular the `database-polling` coordination strategy. When used together with the Elasticsearch backend, the `database-polling` coordination strategy enables distributed applications, as the JMS/JGroups backends in Hibernate Search 5 do, though through a different implementation relying on the database instead of JMS/JGroups queues.

API changes

A lot of APIs changed. We recommend having a look at the getting started guide before migrating.

Annotation mapping

`@Analyzer`

In Hibernate Search 5, it was possible to apply an `@Analyzer` annotation to a class or property, so that the corresponding analyzer would be used by default for any index field declared in this scope.

There is no equivalent to that feature in Hibernate Search 6: all fields must specify their analyzer explicitly using `@FullTextField(analyzer = "myAnalyzer")`, or rely on the (global) default analyzer.

Also, still in Hibernate Search 5, `@Analyzer` could point directly to a class extending `org.apache.lucene.analysis.Analyzer`, for example with `@Analyzer(impl = StandardAnalyzer.class)`.

This is no longer possible: analyzers are now always referenced by their name. However, you can assign a name to a given analyzer instance using the Lucene analysis configurer.

`@AnalyzerDef`, `@AnalyzerDefs`

Annotation-based analyzer definitions are no longer supported.

Instead, implement an analysis configurer: see here for Lucene, or here for Elasticsearch.

`@AnalyzerDiscriminator`

`@AnalyzerDiscriminator` has no direct equivalent in Hibernate Search 6: the analyzer assigned to each field is static and cannot change at runtime, because that results in unreliable matches and in scoring issues.

Instead, Hibernate Search 6 allows declaring multiple index fields for a single property, and putting the content of that property in a different field depending on a discriminator. Then, when searching, you can target all fields at once.

See Mapping multiple alternatives.

`@Boost`

Index-time boosting was deprecated in Hibernate Search 5. It is no longer available in Hibernate Search 6.

Instead, rely on query-time boosting.

`@CacheFromIndex`

This annotation was deprecated and non-functional in Hibernate Search 5. It is no longer available in Hibernate Search 6.

`@CalendarBridge`

`@CalendarBridge` is not necessary to index `Calendar` values: you can simply apply `@GenericField` to a property of type `Calendar`, and an appropriate default bridge will be used.

The main purpose of `@CalendarBridge` in Hibernate Search 5 was to provide the ability to "truncate" calendars upon indexing, e.g. zeroing out all data more precise than the day with `@CalendarBridge(resolution = Resolution.DAY)`.

For such use case, the recommended approach in Hibernate Search 6 is to index values with full resolution (not using `@CalendarBridge`) and to control resolution when searching, with a `range` predicate. Note that you can pass `ZonedDateTime` values to the predicate, which are much easier to truncate manually. For example, to match only documents whose calendar is within a given day:

```
Calendar toMatch = /* ... */;
ZonedDateTime toMatchZonedDateTime = ( (GregorianCalendar) toMatch ).toZonedDateTime();
ZonedDateTime dayStart = toMatchZonedDateTime.truncatedTo( ChronoUnit.DAYS );
ZonedDateTime nextDayStart = dayStart.plus( 1, ChronoUnit.DAYS );

List<MyEntity> hits = searchSession.search( MyEntity.class )
        .where( f -> f.range().field( "calendar" )
                .range( Range.canonical( dayStart, nextDayStart ), ValueConvert.NO )
        .fetchHits( 20 );
```

If that approach doesn't work for you, let us know and we'll try to come up with a solution together.

`@CharFilterDef`

See `@AnalyzerDef`, `@AnalyzerDefs` or `@NormalizerDef`, `@NormalizerDefs`.

`@ClassBridge`, `@ClassBridges`

See Bridges.

`@ContainedIn`

`@ContainedIn` is no longer necessary in Hibernate Search 6.

Hibernate Search 6 infers indexing dependencies from the mapping, and raises errors at bootstrap when the equivalent of `@ContainedIn` cannot be applied automatically (for example an

`@IndexedEmbedded` association with no inverse side).

Thus, the recommended approach when migrating is to simply remove all `@ContainedIn` annotations, then deal with the bootstrap errors, if any.

Hibernate Search 6 is able to raise multiple mapping errors during a single startup, so you don't have to restart the application 20 times to address 20 different problems.

See this section for guidance on how to address these errors.

`@DateBridge`

`@DateBridge` is not necessary to index `Date` values: you can simply apply `@GenericField` to a property of type `java.util.Date`, `java.sql.Date`, `java.sql.Time` or `java.sql.Timestamp`, and an appropriate default bridge will be used.

The main purpose of `@DateBridge` in Hibernate Search 5 was to provide the ability to "truncate" dates upon indexing, e.g. zeroing out all data more precise than the day with `@DateBridge(resolution = Resolution.DAY)`.

For such use case, the recommended approach in Hibernate Search 6 is to index values with full resolution (not using `@DateBridge`) and to control resolution when searching, with a `range` predicate. Note that you can pass `Instant` values to the predicate, which are much easier to truncate manually. For example, to match only documents whose date is within a given day:

```
Date toMatch = /* ... */;
ZoneId userZoneId = /* ... */;
ZonedDateTime toMatchZonedDateTime = Instant.ofEpochMilli( toMatch.getTime() ).atZone(
userZoneId );
ZonedDateTime dayStart = toMatchZonedDateTime.truncatedTo( ChronoUnit.DAYS );
ZonedDateTime nextDayStart = dayStart.plus( 1, ChronoUnit.DAYS );

List<MyEntity> hits = searchSession.search( MyEntity.class )
        .where( f -> f.range().field( "date" )
                .range( Range.canonical( dayStart.toInstant(), nextDayStart.toInstant() ),
                        ValueConvert.NO )
        .fetchHits( 20 );
```

If that approach doesn't work for you, let us know and we'll try to come up with a solution together.

`@DocumentId`

`@DocumentId` is still available in Hibernate Search 6, but moved to a different package: `org.hibernate.search.mapper.pojo.mapping.definition.annotation.DocumentId`.

However, it no longer exposes a `name` attribute, because the document ID is no longer an index field, and thus it does not need a name.

`@DynamicBoost`

Index-time boosting was deprecated in Hibernate Search 5. It is no longer available in Hibernate Search 6.

Instead, rely on query-time boosting.

`@Facet`, `@Facets`

Basics

Facets are now called aggregations, which are a generalization of the concept of faceting.

To make a field aggregable, just set the `@*Field.aggregable` attribute to `Aggregable.YES`:

```
@KeywordField(aggregable = Aggregable.YES)
private String myKeyword;
@GenericField(aggregable = Aggregable.YES)
private Integer myInteger;
@GenericField(aggregable = Aggregable.YES)
private LocalDate myLocalDate;
```

An aggregable `@KeywordField` with a normalizer will return normalized values in aggregations, whereas Hibernate Search 5 used to return raw (non-normalized) values.

See Faceting returns normalized strings.

`aggregable` is not available on `@FullTextField`, because aggregation on a tokenized field would aggregate tokens instead of field values, which is rarely the intent.

If you need both an analyzer and aggregations on the same property, create two separate fields:

```
@FullTextField
@KeywordField(name = "category_aggregation", aggregable = Aggregable.YES)
private String category;
```

For instructions on how to execute aggregations, see Facets ⇒ Aggregations.

This will lead to two separate fields being created in the index, for the same property. Just make sure to use the correct field name when searching: `category` when creating predicates, but `category_aggregation` when creating aggregations.

`@Facet.encoding`

The facet encoding options no longer exists: strings will be indexed as strings and numbers will be indexed as numbers.

For the few cases where encoding a number as a string is necessary, you can define a separate field exclusively for aggregations, and apply a custom value bridge to convert the number to a string (and back).

`@Field`, `@Fields`

Basics

The `@Field` annotation was split into multiple annotations, specific to each field type:

- `org.hibernate.search.mapper.pojo.mapping.definition.annotation.FullTextField`
- `org.hibernate.search.mapper.pojo.mapping.definition.annotation.KeywordField`
- `org.hibernate.search.mapper.pojo.mapping.definition.annotation.GenericField`

Here is a quick reference of how to convert a `@Field` annotation to Hibernate Search 6:

Property type	Hibernate Search 5	Hibernate Search 6
`String`, `Character`, `char`, enum	`@Field`	`@FullTextField`
	`@Field(analyzer = @Analyzer (definition = "myAnalyzer"))`	`@FullTextField(analyzer = "myAnalyzer")`
	`@Field @Analyzer(definition = " myAnalyzer")`	
	`@Field(normalizer = @Normalizer (definition = "myNormalizer"))`	`@KeywordField(normalizer = "myNormalizer")`

	`@Field(analyze = analyze.NO)`	`@KeywordField`
		or
		`@GenericField`

Other	`@Field @NumericField`	`@GenericField`
	`@Field`	`@GenericField`

`@Field.analyze`

`@Field.analyze` has no direct equivalent in Hibernate Search 6. Instead of enabling/disabling analysis explicitly, pick the right `@*Field` annotation according to your needs.

`@Field.analyzer`

See Basics.

`@Field.boost`

Index-time boosting was deprecated in Hibernate Search 5. It is no longer available in Hibernate Search 6.

Instead, rely on query-time boosting.

`@Field.bridge`

See Bridges.

`@Field.index`

`@Field.index` is now `@*Field.searchable`:

Hibernate Search 5	Hibernate Search 6

`@Field(index = Index.YES)`	`@GenericField(searchable = Searchable.YES)`
	or
	`@FullTextField(searchable = Searchable.YES)`
	or
	`@KeywordField(searchable = Searchable.YES)`

`@Field(index = Index.NO)`	`@GenericField(searchable = Searchable.NO)`
	or
	`@FullTextField(searchable = Searchable.NO)`
	or
	`@KeywordField(searchable = Searchable.NO)`

`@Field.indexNullAs`

`@Field.indexNullAs` is still available for most Hibernate Search 6's `@*Field` annotations:

Hibernate Search 5	Hibernate Search 6

`@Field(indexNullAs = "_null_")`	`@GenericField(indexNullAs = "_null_")`
	or
	`@KeywordField(indexNullAs = "_null_")`

However:

- You should consider whether it is really necessary, as the new `exists` predicate introduced in Hibernate Search 6 allows finding documents where a field is present or not **without** relying on `indexNullAs`.
- `indexNullAs` is not available on `@FullTextField`.
- The default null token is no longer supported, i.e. `Field.DEFAULT_NULL_TOKEN` has no equivalent in Hibernate search 6. Each field that requires `indexNullAs` must have its value set explicitly.
- The (text) value passed to `indexNullAs` must be formatted according to the type of the field.
- `indexNullAs` is irrelevant when building predicates and when projecting.

`@Field.name`

`@Field.name` stays the same in Hibernate Search 6's `@*Field` annotations:

Hibernate Search 5	Hibernate Search 6

`@Field(name = "myField")`	`@GenericField(name = "myField")`
	or
	`@FullTextField(name = "myField")`
	or
	`@KeywordField(name = "myField")`

`@Field.normalizer`

See Basics.

`@Field.norms`

`@Field.norms` only has an equivalent in Hibernate Search 6's `@FullTextField` and `@KeywordField`. The `Norms` enum has moved to `org.hibernate.search.engine.backend.types.Norms`.

`@Field.store`

`@Field.store` is now `@*Field.projectable`:

Hibernate Search 5	Hibernate Search 6

`@Field(store = Store.YES)`	`@GenericField(projectable = Projectable.YES)`
	or
	`@FullTextField(projectable = Projectable.YES)`
	or
	`@KeywordField(projectable = Projectable.YES)`

`@Field(store = Store.NO)`	`@GenericField(projectable = Projectable.NO)`
	or
	`@FullTextField(projectable = Projectable.NO)`
	or
	`@KeywordField(projectable = Projectable.NO)`

`@Field(store = Store.COMPRESS)`	No direct equivalent; use `Projectable.YES`. See also HSEARCH-3081.

`@Field.termVector`

`@Field.termVector` only has an equivalent in Hibernate Search 6's `@FullTextField`. The `TermVector` enum has moved to `org.hibernate.search.engine.backend.types.TermVector`.

`@FieldBridge`

See Bridges.

`@FullTextFilterDef`, `@FullTextFilterDefs`

Full-text filters have no direct equivalent in Hibernate Search 6.

See Full-text filter.

`@Indexed`

Basics

`@Indexed` is still available in Hibernate Search 6, but moved to a different package: `org.hibernate.search.mapper.pojo.mapping.definition.annotation.Indexed`.

However, in Hibernate Search 6:

- index names default to the entity name, not the class name.
- `@Indexed` is inherited.
- each indexed type has its own index.

`@Indexed.index`

`@Indexed.index` stays the same in Hibernate Search 6.

`@Indexed.interceptor`

Entity indexing interceptors have no direct equivalent in Hibernate Search 6, but conditional indexing can be implemented through routing bridges.

See Entity indexing interceptors.

`@IndexedEmbedded`

Basics

`@IndexedEmbedded` is still available in Hibernate Search 6, but moved to a different package: `org.hibernate.search.mapper.pojo.mapping.definition.annotation.IndexedEmbedded`.

`@IndexedEmbedded.depth`

`@IndexedEmbedded.depth` was renamed to `includeDepth` in Hibernate Search 6:

Hibernate Search 5	Hibernate Search 6
`@IndexedEmbedded(depth = 2)`	`@IndexedEmbedded(includeDepth = 2)`
`@IndexedEmbedded(depth = 1, includePaths = {"foo.bar", "foo.bar2"})`	`@IndexedEmbedded(includeDepth = 1, includePaths = {"foo.bar", "foo.bar2"})`

`@IndexedEmbedded.includePaths`

`@IndexedEmbedded.includePaths` stays the same in Hibernate Search 6.

However, the document id of other entities is no longer a field by default, so you can no longer use `@IndexedEmbedded(includePaths = "id")` (for example) to embed another entity's ID, unless you explicitly add a `@GenericField` annotation on the id property. See The document ID is not an index field for more information.

`@IndexedEmbedded.prefix`

`@IndexedEmbedded.prefix` is still available in Hibernate Search 6, but is deprecated for removal in the next major version.

You should use `@IndexedEmbedded.name` instead, which doesn't prepend a prefix to the embedded fields, but instead creates an object field with the given name:

Hibernate Search 5	Hibernate Search 6
`@IndexedEmbedded(prefix = "foo.")`	`@IndexedEmbedded(name = "foo")`
`@IndexedEmbedded(prefix = "foo.bar.")`	⚠ No equivalent: `name` only allows one object field.
`@IndexedEmbedded(prefix = "foo_")`	⚠ No equivalent: `name` does not allow prefixes to be prepended to embedded field names.
`@IndexedEmbedded(prefix = "foo.bar_")`	

`@IndexedEmbedded.targetElement`

`@IndexedEmbedded.targetElement` was renamed to `targetType` in Hibernate Search 6:

Hibernate Search 5	Hibernate Search 6
`@IndexedEmbedded(targetElement = MyConcreteEntity.class)`	`@IndexedEmbedded(targetType = MyConcreteEntity.class)`

`@IndexedEmbedded.indexNullAs`

`@IndexedEmbedded.indexNullAs` has no equivalent in Hibernate Search 6.

To search for documents where an object field is present (or absent), use the `exists` predicate.

`@IndexedEmbedded.includeEmbeddedObjectId`

`@IndexedEmbedded.includeEmbeddedObjectId` stays the same in Hibernate Search 6.

However, embedded IDs of numeric or date/time types (`Integer`, `Long`, `Date`, ...) used to be indexed as string values by default in Hibernate Search 5, but are indexed as numeric values by default in Hibernate Search 6. See More fields are numeric by default.

Using `@IndexedEmbedded` to request container extraction

A little-known and undocumented feature of `@IndexedEmbedded` was to combine it with `@Field` on a property of a container type (`Collection`, `List`, `Map`, ...) to instruct Hibernate Search to apply `@Field` to the container elements instead of the container.

For example, the code below would lead to a bootstrap failure, because there is no default bridge for the `List` type:

```
@Field
@ElementCollection
private List<String> notes;
```

However, the code below would work just fine, and would instruct Hibernate Search to index each element of the `List` in the `notes` index field:

```
@Field
@IndexedEmbedded
@ElementCollection
private List<String> notes;
```

In Hibernate Search 6, `@IndexedEmbedded` should no longer be used this way, as the container extraction is now implied:

```
@FullTextField
@ElementCollection
private List<String> notes;
```

See also Container extraction for `@*Field` is implied.

`@Key`

`@Key` has no equivalent in Hibernate Search 6.

See also Full-text filter.

`@Latitude`

`@Latitude` is still available in Hibernate Search 6, but moved to a different package: `org.hibernate.search.annotations.Latitude`.

See also `@Spatial`, `@Spatials`.

`@Longitude`

`@Longitude` is still available in Hibernate Search 6, but moved to a different package: `org.hibernate.search.annotations.Longitude`.

See also `@Spatial`, `@Spatials`.

`@Normalizer`

In Hibernate Search 5, `@Normalizer` could point directly to a class extending `org.apache.lucene.analysis.Analyzer`, for example with `@Normalizer(impl = MyNormalizer.class)`.

This is no longer possible: normalizers are now always referenced by their name. However, you can assign a name to a given normalizer instance using the Lucene analysis configurer.

`@NormalizerDef`, `@NormalizerDefs`

Annotation-based normalizer definitions are no longer supported.

Instead, implement an analysis configurer: see here for Lucene, or here for Elasticsearch.

`@NumericField`, `@NumericFields`

`@NumericField` no longer exists in Hibernate Search 6.

Numeric types are indexed as numeric values by default, so this annotation can simply be removed.

See also More fields are numeric by default.

`@ProvidedId`

`@ProvidedId` was deprecated in Hibernate Search 5. It no longer exists in Hibernate Search 6.

`@SortableField`, `@SortableFields`

`@SortableField` no longer exists in Hibernate Search 6. Instead, use `@*Field.sortable`:

Hibernate Search 5	Hibernate Search 6

`@Field`
`@SortableField`
	`@GenericField(sortable = Sortable.YES)`
	
	or
	
	`@KeywordField(sortable = Sortable.YES)`

`@Field`
`@Field(name = "myField_sort", analyze = Analyze.NO)`
`@SortableField(forField = "myField_sort")`
	`@FullTextField`
	`@KeywordField(name = "myField_sort", sortable = Sortable.YES)`

`sortable` is not available on `@FullTextField`, because tokenized data cannot be reliably sorted on.

If you need both an analyzer and sorts on the same property, create two separate fields:

```
@FullTextField
@KeywordField(name = "title_sort", normalizer = "myNormalizer", sortable = Sortable.YES)
private String title;
```

This will lead to two separate fields being created in the index, for the same property. Just make sure to use the correct field name when searching: `title` when creating predicates, but `title_sort` when creating sorts.

`@Spatial`, `@Spatials`

Basics

`@Spatial` has no direct equivalent in Hibernate Search 6.

Here is a quick reference of how to convert a `@Spatial` annotation to Hibernate Search 6:

<table>
<tr><th>Hibernate Search 5</th><th>Latitude/longitude are mutable?</th><th>Hibernate Search 6</th></tr>
<tr><td rowspan="2">

```
@Indexed
@Entity
public class MyEntity {
    @Spatial
    private MyCoordinates
location;
}
public class MyCoordinates
implements Coordinates {
    private Double latitude;
    private Double longitude;
    @Override
    public Double getLatitude() {
        return latitude;
    }
    @Override
    public Double getLongitude() {
        return longitude;
    }
}
```

</td><td>Yes</td><td>

```
@Indexed
@Entity
public class MyEntity {
    @GeoPointBinding(projectable =
Projectable.YES)
    private MyCoordinates
location;
}
public class MyCoordinates {
    private Double latitude;
    private Double longitude;
    @Latitude
    public Double getLatitude() {
        return latitude;
    }
    @Longitude
    public Double getLongitude() {
        return longitude;
    }
}
```

</td></tr>
<tr><td>No</td><td>

```
@Indexed
@Entity
public class MyEntity {
    @GenericField(projectable =
Projectable.YES)
    private MyCoordinates
location;
}
public class MyCoordinates
implements GeoPoint {
    private final double latitude;
    private final double
longitude;
    @Override
    public double latitude() {
        return latitude;
    }
    @Override
    public double longitude() {
        return longitude;
    }
}
```

</td></tr>
</table>

Hibernate Search 5	Latitude/longitude are mutable?	Hibernate Search 6

`@Indexed`
`@Entity`
`@Spatial`
`public class MyEntity {`
`  @Latitude`
`  private Double latitude;`
 	
`  @Longitude`
`  private Double longitude;`
`}`
	Yes	`@Indexed`
		`@Entity`
		`@GeoPointBinding(projectable = Projectable.YES)`
		`public class MyEntity {`
		`  @Latitude`
		`  private Double latitude;`
		
		`  @Longitude`
		`  private Double longitude;`
		`}`

`@Indexed`
`@Entity`
`@Spatial(name = "home_coordinates")`
`@Spatial(name = "work_coordinates")`
`public MyEntity {`
`  @Latitude(of = "home_coordinates")`
`  private Double homeLatitude;`
 	
`  @Longitude(of = "home_coordinates")`
`  private Double homeLongitude;`
 	
`  @Latitude(of = "work_coordinates")`
`  private Double workLatitude;`
 	
`  @Longitude(of = "home_coordinates")`
`  private Double workLongitude;`
`}`
	Yes	`@Indexed`
		`@Entity`
		`@GeoPointBinding(name = "home_coordinates", markerSet = "home", projectable = Projectable.YES)`
		`@GeoPointBinding(name = "work_coordinates", markerSet = "work", projectable = Projectable.YES)`
		`public MyEntity {`
		`  @Latitude(markerSet = "home")`
		`  private Double homeLatitude;`
		
		`  @Longitude(markerSet = "home")`
		`  private Double homeLongitude;`
		
		`  @Latitude(markerSet = "work")`
		`  private Double workLatitude;`
		
		`  @Longitude(markerSet = "home")`
		`  private Double workLongitude;`
		`}`

`@Spatial.boost`

Index-time boosting was deprecated in Hibernate Search 5. It is no longer available in Hibernate Search 6.

Instead, rely on query-time boosting.

`@Spatial.name`

`@Field.name` stays the same in Hibernate Search 6's `@GeoPointBinding` and `@GenericField` annotations, with one exception: the field name is mandatory when `@GeoPointBinding` is applied to a class.

Hibernate Search 5	Hibernate Search 6

`@Spatial(name = "myField")`	`@GeoPointBinding(name = "myField")`
	or
	`@GenericField(name = "myField")`

`@Spatial`
`public class MyEntity {`
`  // ...`
`}`
	`@GeoPointBinding(name = "location")`
	`public class MyEntity {`
	`  // ...`
	`}`

`@Spatial.spatialMode`, `@Spatial.topSpatialHashLevel`, `@Spatial.bottomSpatialHashLevel`

`@Spatial.spatialMode`, `@Spatial.topSpatialHashLevel`, `@Spatial.bottomSpatialHashLevel` have no equivalent in Hibernate Search 6: geohash-based geo-point fields are no longer supported.

For Lucene, this is because Hibernate Search now uses Lucene's built-in spatial support, which is range-based.

For Elasticsearch, hash-based geo-point fields have never been available in the first place.

`@Spatial.store`

`@Spatial.store` is now `@GeoPointBinding.projectable`/`@GenericField.projectable`:

Hibernate Search 5	Hibernate Search 6

`@Spatial(store = Store.YES)`	`@GeoPointBinding(projectable = Projectable.YES)`
	or
	`@GenericField(projectable = Projectable.YES)`

Hibernate Search 5	Hibernate Search 6

`@Spatial(store = Store.NO)`	`@GeoPointBinding(projectable = Projectable.YES)`
	
	or
	
	`@GenericField(projectable = Projectable.YES)`

In Hibernate Search 6, `projectable` must be set to `Projectable.YES` to enable distance projections on a geo-point field.

`@TikaBridge`

`@TikaBridge` has no equivalent in Hibernate Search 6 yet.

If you need this feature, vote for it to be re-implemented using the new bridge API in a later version of Hibernate Search: HSEARCH-3350. We'll also gladly help anyone interested in contributing a patch: feel free to contact us.

`@TokenFilterDef`

See `@AnalyzerDef`, `@AnalyzerDefs` or `@NormalizerDef`, `@NormalizerDefs`.

`@TokenizerDef`

See `@AnalyzerDef`, `@AnalyzerDefs`.

Programmatic mapping

Programmatic mapping was overhauled to match the new mapping annotations.

See here for the entry point, and refer to the javadoc for details.

Analysis definition provider

Analysis definition providers are now called analysis configurers. The interfaces are slightly different but follow the same general principle.

See here for Lucene, or here for Elasticsearch.

Analysis configurers can be used to override the default analyzer. To do so, just define a custom analyzer named `default`.

With the Lucene backend, analysis configurers can be used to override the default similarity. See here for more information

Bridges

The bridge API was completely reworked in Hibernate Search 6 to offer a more powerful, Lucene-independent solution. New features include:

- the ability to define field types precisely, allowing in particular to pick an analyzer or to enable aggregation (faceting) on a bridge-declared field;
- the ability to declare the properties the bridge relies on, allowing Hibernate Search to reindex less often;
- the ability to declare dynamic fields with a precise type which the Search DSL will be aware of;
- the ability to define custom field annotations;
- and more.

If your application relied on custom bridges with Hibernate Search 5, and you need to re-implement them with Hibernate Search 6, see mapping custom property types.

The new bridge API is quite different from Hibernate Search 5, but most changes should be addressed rather easily, especially for the simpler bridges which will be implemented through `ValueBridge`.

Perhaps the only change to note is that bridges must declare the index fields they will create at bootstrap. To set the value of new fields with random names dynamically at runtime, you will have to rely on field templates.

Entity indexing interceptors

Entity indexing interceptors were used in Hibernate Search 5 to implement conditional indexing.

Conditional indexing can be implemented in Hibernate Search 6 with a different, but equivalent solution: the `RoutingBridge`.

The idea is for the `RoutingBridge` to decide, based on the state of an entity, whether its document should be routed to the index (indexed) or not (not indexed).

See this section of the documentation for more information.

`FullTextEntityManager`/`FullTextSession`

Basics

The equivalent to Hibernate Search 5's `FullTextEntityManager`/`FullTextSession` is Hibernate Search 6's `SearchSession`.

Here is how to retrieve the `FullTextEntityManager`/`FullTextSession` in Hibernate Search 5:

```
FullTextEntityManager fullTextEntityManager = Search.getFullTextEntityManager( entityManager
);
FullTextSession fullTextSession = Search.getFullTextSession( session );
```

The main entry point to Hibernate Search APIs is still named `Search`, but it moved to another package: `org.hibernate.search.mapper.orm.Search`. Here is how to retrieve the `SearchSession` in Hibernate Search 6:

```
SearchSession searchSession = Search.session( entityManager );
// OR
SearchSession searchSession = Search.session( session );
```

Tha main difference between Hibernate Search 5 and 6 is that `SearchSession` does not extend `EntityManager` or `Session`: the interface stands on its own.

To retrieve the `EntityManager`/`Session` for a given `SearchSession`, you can simply call `searchSession.toEntityManager()`/`searchSession.toOrmSession()`.

See the following sections for the exact equivalent of each operation.

`FullTextSession.createFullTextQuery`

See Searching.

`FullTextSession.index(…)`, `FullTextSession.purge(…)`

In Hibernate Search 6, indexing operations are handled through the indexing plan:

- `fullTextSession.index(entity)` is equivalent to `searchSession.indexingPlan().addOrUpdate(entity)`;
- `fullTextSession.purge(entity)` is equivalent to `searchSession.indexingPlan().delete(entity)`;
- `fullTextSession.purge(entityType, id)` is equivalent to `searchSession.indexingPlan().purge(entityType, id, null)` (if you don't use

sharding) or `searchSession.indexingPlan().purge(entityType, id, routingKey)` (if you do use sharding).

However:

- In Hibernate Search 5, these operations used to bypass the custom Entity indexing interceptors. In Hibernate Search 6, the `addOrUppdate` and `delete` operations do not bypass the equivalent `RoutingBridge`, but the `purge` operation still does.
- In Hibernate Search 5, `index` only triggered indexing of the entity passed as an argument. In Hibernate Search 6, `addOrUpdate` will also trigger reindexing of other entities that embed it (through `@IndexedEmbedded` for example).
- In Hibernate Search 5, passing a `null` identifier to `purge` used to trigger removal of all entities from the index. In Hibernate Search 6, passing `null` to `delete`/`purge` will result in an exception being thrown. To remove all entities from the index, see `FullTextSession.purgeAll(…)`.

See this section of the documentation for more information.

`FullTextSession.flushToIndexes()`

The equivalent of `fullTextSession.flushToIndexes()` is `searchSession.indexingPlan().execute()`.

However, keep in mind that Hibernate Search 6 (on contrary to 5) turns entities into documents upon Hibernate ORM session flushes, so you generally don't need to write to indexes until the write happens automatically on transaction commit, **even if you manually flush/clear the Hibernate ORM session**.

The only valid reason to use this method is to preserve memory when you're processing a very large number of entities.

See this section of the documentation for more information.

`FullTextSession.purgeAll(…)`

In Hibernate Search 5:

```
FullTextSession ftSession = Search.getFullTextSession( session );

ftSession.purgeAll( Book.class );
```

Equivalent in Hibernate Search 6:

```
SearchSession searchSession = Search.session( session );

searchSession.workspace( Book.class ).purge();
```

However, the purge is no longer performed on transaction commit: it's executed immediately.

See this section of the documentation for more information.

`FullTextSession.createIndexer(…)`

See `MassIndexer`.

Searching

`FullTextQuery` ⇒ DSL

Basics

Search APIs have changed significantly, in order to implement several improvements, in particular:

- to get rid of Lucene types leaking through Hibernate Search APIs;
- to avoid returning raw types in search results;
- to expose a more adapted, native interface for Hibernate Search queries, instead of trying to make do with JPA's `Query` type;
- to offer a less verbose, lambda-based syntax as an alternative to the "traditional", object-based syntax.

The recommended way to build search queries in Hibernate Search 6 is through the Hibernate Search DSL. You can find an explanation of entry points and all available features of this DSL in the dedicated section of the documentation.

As to migrating existing queries, let's take the following query in Hibernate Search 5 as an example:

```
int pageIndex = ...;
int pageSize = ...;
FullTextEntityManager fullTextEntityManager = Search.getFullTextEntityManager( em );
QueryBuilder qb = fullTextEntityManager.getSearchFactory()
        .buildQueryBuilder().forEntity( Book.class ).get();

org.apache.lucene.search.Query luceneQuery = qb.keyword()
        .onField( "title" ).boostedTo( 3 )
        .matching( "robot" )
        .createQuery();

FullTextQuery fullTextQuery = fullTextEntityManager.createFullTextQuery( luceneQuery, Book
.class );
fullTextQuery.setFirstResult( pageIndex * pageSize );
fullTextQuery.setMaxResults( pageSize );

List hits = fullTextQuery.getResultList();
int totalHitCount = fullTextQuery.getResultSize();
```

With the recommended lambda-based syntax, the equivalent code in Hibernate Search 6 will be:

```
int pageIndex = ...;
int pageSize = ...;
SearchSession session = Search.session( entityManager );

SearchResult<Book> result = session.search( Book.class )
        .where( f -> f.match().field( "title" ).matching( "robot" ).boost( 3 ) )
        .fetch( pageIndex * pageSize, pageSize );

List<Book> hits = result.hits();
long totalHitCount = result.total().hitCount();
```

Alternatively, if the total hit count is not desired, you can use `fetchHits()`:

```
int pageIndex = ...;
int pageSize = ...;
SearchSession session = Search.session( entityManager );

List<Book> hits = session.search( Book.class )
        .where( f -> f.match().field( "title" ).matching( "robot" ).boost( 3 ) )
        .fetchHits( pageIndex * pageSize, pageSize );
```

Adapter from search query to JPA/ORM query

If you really need a `Query` object implementing JPA or Hibernate ORM interfaces, for example to integrate with external code designed for JPA/Hibernate ORM, know that the Hibernate Search query can still be converted.

See this section of the documentation for more information.

`FullTextQuery.initializeObjectsWith`

Hibernate Search 6 does not allow setting a `DatabaseRetrievalMethod` on a search query: entities are always loaded with a Hibernate ORM query.

In Hibernate Search 6, the equivalent to setting the `ObjectLookupMethod` with `FullTextQuery.initializeObjectsWith` is to set the cache lookup strategy.

`FullTextQuery.setCriteriaQuery(…)`

Hibernate Search 6 does not allow adding a `Criteria` object to a search query.

If your goal is to control loading of associations precisely, set an entity graph in loading options instead.

If your goal is to apply a filter expressed by an SQL "where" clause executed in-database, rework your query to project on the entity ID, and execute a JPA/Hibernate ORM query after the search query to filter the entities and load them.

`FullTextQuery.explain(…)`

Hibernate Search 6 still offers a way to explain the score of hits through an `explain` method, but that method expects the entity ID, not the internal Lucene document ID (which can change from one query execution to the next).

```
LuceneSearchQuery<Book> query = searchSession.search( Book.class )
        .extension( LuceneExtension.get() )
        .where( f -> f.match()
                .field( "title" )
                .matching( "robot" ) )
        .toQuery();

Explanation explanation = query.explain( 1 );
```

See this section of the documentation.

`FullTextQuery.setFilter(…)`

`FullTextQuery.setFilter(…)` was deprecated in Hibernate Search 5. It is no longer available in Hibernate Search 6.

To filter a query, just wrap your predicate in a boolean predicate and add a `filter` clause.

`FullTextQuery.enableFullTextFilter(…)` / `FullTextQuery.disableFullTextFilter(…)`

See Full-text filter.

`org.apache.lucene.search.Query` ⇒ `SearchPredicate`

Basics

Lucene queries are replaced with Lucene-independent "search predicates" in Hibernate Search 6.

Most of the time, code that builds queries does not need to manipulate search predicates directly, thanks to the lambda syntax. However, it's still possible to manipulate `SearchPredicate` objects if you need to pass them around from a method to another.

You can find more information about building predicates and details about all available predicates in the dedicated section of the documentation, and instructions to migrate from the Hibernate Search 5 Query DSL in `Query` DSL migration reference.

As to migrating existing complex queries, let's consider the query below:

```
MySearchParameters searchParameters = ...;
FullTextEntityManager fullTextEntityManager = Search.getFullTextEntityManager( em );
QueryBuilder qb = fullTextEntityManager.getSearchFactory()
        .buildQueryBuilder().forEntity( Book.class ).get();

BooleanJunction junction = qb.bool();
junction.must(qb.all().createQuery());

if ( searchParameters.getSearchTerms() != null ) {
    junction.must( qb.simpleQueryString().onFields( "title", "description" )
            .withAndAsDefaultOperator()
            .matching( searchParameters.getSearchTerms() )
            .createQuery() );
}
if ( searchParameters.getMaxBookLength() != null ) {
    junction.must( qb.range().onField( "pageCount" )
            .below( searchParameters.getMaxBookLength() ) );
}
if ( !searchParameters.getGenres().isEmpty() ) {
    BooleanJunction junction2 = qb.bool();
    for ( Genre genre : searchParameters.getGenres() ) {
        junction2.should( qb.keyword().onField( "genre" )
                .matching( genre ) );
    }
    junction.must( junction2.createQuery() );
}

org.apache.lucene.search.Query luceneQuery = junction.createQuery();

FullTextQuery fullTextQuery = fullTextEntityManager.createFullTextQuery( luceneQuery, Book
.class );
fullTextQuery.setFirstResult( params.getPageIndex() * params.getPageSize() );
fullTextQuery.setMaxResults( params.getPageSize() );

List hits = fullTextQuery.getResultList();
```

It would look like this in Hibernate Search 6:

```
MySearchParameters searchParameters = ...;
SearchSession session = Search.session( entityManager );
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.bool( b -> {
            b.must( f.matchAll() );
            if ( searchParameters.getSearchTerms() != null ) {
                b.must( f.simpleQueryString().fields( "title", "description" )
                        .matching( searchParameters.getSearchTerms() )
                        .defaultOperator( BooleanOperator.AND ) );
            }
            if ( searchParameters.getMaxBookLength() != null ) {
                b.must( f.range().field( "pageCount" )
                        .atMost( searchParameters.getMaxBookLength() ) );
            }
            if ( !searchParameters.getGenres().isEmpty() ) {
                b.must( f.bool( b2 -> {
                    for ( Genre genre : searchParameters.getGenres() ) {
                        b2.should( f.match().field( "genre" )
                                .matching( genre ) );
                    }
                } ) );
            }
        } ) )
        .fetchHits( searchParameters.getPageIndex() * searchParameters.getPageSize(),
                searchParameters.getPageSize() );
```

Alternatively, if for some reasons predicate objects are necessary:

```
MySearchParameters searchParameters = ...;
SearchSession session = Search.session( entityManager );
SearchPredicateFactory pf = session.scope( Book.class ).predicate();
List<SearchPredicate> predicates = new ArrayList<>();

if ( searchParameters.getSearchTerms() != null ) {
    predicates.add( pf.simpleQueryString().fields( "title", "description" )
            .matching( searchParameters.getSearchTerms() )
            .defaultOperator( BooleanOperator.AND )
            .toPredicate() );
}
if ( searchParameters.getMaxBookLength() != null ) {
    predicates.add( pf.range().field( "pageCount" )
            .atMost( searchParameters.getMaxBookLength() )
            .toPredicate() );
}
if ( !searchParameters.getGenres().isEmpty() ) {
    predicates.add( f.bool( b -> {
        for ( Genre genre : searchParameters.getGenres() ) {
            b.should( f.match().field( "genre" )
                    .matching( genre ) );
        }
    } )
            .toPredicate() );
}

SearchPredicate topLevelPredicate = pf.bool( b -> {
    b.must( f.matchAll() );
    for ( SearchPredicate predicate : predicates ) {
        b.must( predicate );
    }
} )
        .toPredicate();

List<Book> hits = searchSession.search( Book.class )
        .where( topLevelPredicate )
        .fetchHits( searchParameters.getPageIndex() * searchParameters.getPageSize(),
                searchParameters.getPageSize() );
```

`Query` DSL migration reference

The code below makes some assumptions:

- For Hibernate Search 5, a `QueryBuilder` was retrieved from the `SearchFactory` and put in variable `qb`.
- For Hibernate Search 6, the predicate is being built in a lambda expression: `Search.session( entityManager ).search( Book.class ).where( f → … )`.

Hibernate Search 5	Hibernate Search 6	Documentation
`qb.all().createQuery()`	`f.matchAll()`	`matchAll`

Hibernate Search 5	Hibernate Search 6	Documentation

`qb.keyword().onField("field")`
`.matching("value")`
`.createQuery()`
	`f.match().field("field")`
	`.matching("value")`
		`match`
		
		For matches on the identifier (which is no longer a field by default), use the `id` predicate instead (see below).
		
		`matching(…)` no longer accepts `null`; use a (negated) `exists` predicate instead (see below).

`qb.keyword().onField("field")`
`.matching("value")`
`.ignoreFieldBridge()`
`.createQuery()`
	`f.match().field("field")`
	`.matching("value",`
	
	`ValueConvert.NO)`	

`qb.keyword().onField("field")`
`.matching("value")`
`.ignoreAnalyzer()`
`.createQuery()`
	`f.match().field("field")`
	`.matching("value")`
	`.skipAnalysis()`	

`qb.keyword().onField("field")`
`.fuzzy()`
`.matching("value")`
`.createQuery()`
	`f.match().field("field")`
	`.matching("value")`
	`.fuzzy()`	

`qb.keyword().onField("field")`
`.fuzzy()`
 	
`.withEditDistanceUpTo(2)`
`.matching("value")`
`.createQuery()`
	`f.match().field("field")`
	`.matching("value")`
	`.fuzzy(2)`	

`qb.keyword().onField("field")`
`.fuzzy()`
 	
`.withEditDistanceUpTo(2)`
`.withPrefixLength(1)`
`.matching("value")`
`.createQuery()`
	`f.match().field("field")`
	`.matching("value")`
	`.fuzzy(2,1)`	

`qb.keyword().onField("field")`
`.matching(null)`
`.createQuery()`
	`f.bool().mustNot(f.exists()`
	`.field("field"))`
		`exists`

`qb.keyword().onField("id")`
`.matching(123L)`
`.createQuery()`
	`f.id().matching(123L)`	`id`

`qb.keyword().wildcard()`
`.onField("field")`
`.matching("pattern*")`
`.createQuery()`
	`f.wildcard().field("field")`
	`.matching("pattern*")`
		`wildcard`

Hibernate Search 5	Hibernate Search 6	Documentation

`qb.range().onField("field")`
`.from(0).to(3)`
`.createQuery()`
	`f.range().field("field")`
	`.between(0,3)`
		range

`qb.range().onField("field")`
 	
`.from(0).to(3).excludeLimit()`
`.createQuery()`
	`f.range().field("field")`
	
	`.range(Range.canonical(0,3))`	

`qb.range().onField("field")`
`.below(3)`
`.createQuery()`
	`f.range().field("field")`
	`.atMost(3)`	

`qb.range().onField("field")`
 	
`.below(3).excludeLimit()`
`.createQuery()`
	`f.range().field("field")`
	`.lessThan(3)`	

`qb.range().onField("field")`
`.above(0)`
`.createQuery()`
	`f.range().field("field")`
	`.atLeast(0)`	

`qb.range().onField("field")`
 	
`.above(0).excludeLimit()`
`.createQuery()`
	`f.range().field("field")`
	`.greaterThan(0)`	

`qb.phrase().onField("field")`
`.sentence("a few words")`
`.createQuery()`
	`f.phrase().field("field")`
	`.matching("a few words")`
		phrase

`qb.bool()`
`.must(qb.keyword().onField("field1")`
`.matching("value1")`
`.createQuery())`
`.should(qb.keyword().onField("field2")`
`.matching("value2")`
`.createQuery())`
`.filter(qb.keyword().onField("field3")`
`.matching("value3")`
`.createQuery())`
`.must(qb.keyword().onField("field4")`
`.matching("value4")`
`.createQuery())`
`.not()`
`.createQuery()`
	`f.bool()`
	`.must(f.match().field("field1")`
	`.matching("value1"))`
	`.should(f.match().field("field2")`
	`.matching("value2"))`
	`.filter(f.match().field("field3")`
	`.matching("value3"))`
	`.mustNot(f.match().field("field4")`
	`.matching("value4"))`
		bool

`qb.bool()`
`.minimumShouldMatchNumber(2)`
`.should(qb.keyword().onField("field1")`
`.matching("value1")`
`.createQuery())`
`.should(qb.keyword().onField("field2")`
`.matching("value2")`
`.createQuery())`
`.createQuery()`
	`f.bool()`
	`.minimumShouldMatchNumber(2)`
	`.should(f.match().field("field1")`
	`.matching("value1"))`
	`.should(f.match().field("field2")`
	`.matching("value2"))`	

Hibernate Search 5	Hibernate Search 6	Documentation

`qb.simpleQueryString().onField("field")`
 	
`.matching("querystring")`
	`f.simpleQueryString().field("field")`
	
	`.matching("querystring")`
		`simpleQueryString`

`qb.simpleQueryString()`
`.onField("field")`
 	
`.withAndAsDefaultOperator()`
 	
`.matching("querystring")`
	`f.simpleQueryString().field("field")`
	
	`.defaultOperator(BooleanOperator.AND)`
	
	`.matching("querystring")`	

`qb.spatial().onField("field")`
`.within(51,Unit.KM)`
 	
`.ofLatitude(centerLatitude)`
 	
`.andLongitude(centerLongitude)`
`.createQuery()`
	`f.spatial().within()`
	`.field("field")`
	`.circle(centerLatitude,`
	
	`centerLongitude,`
	`51,`
	
	`DistanceUnit.KILOMETERS)`
		`within`

`Coordinates center = ...;`
`qb.spatial().onField("field")`
`.within(51,Unit.KM)`
`.ofCoordinates(center)`
`.createQuery()`
	`GeoPoint center = ...;`
	`f.spatial().within()`
	`.field("field")`
	`.circle(center,`
	`51,`
	
	`DistanceUnit.KILOMETERS)`	

`qb.moreLikeThis()`	–	No equivalent in Hibernate Search 6. If you need more-like-this predicates, feel free to drop a comment to explain your use case on HSEARCH-3272.

Native query

It is still possible to rely on native Lucene queries (e.g. `new RegexpQuery(…)`) or Elasticsearch queries (e.g. `{'match': {…}}`) in Hibernate Search 6: you will just need to rely on the backend-specific extension.

Be aware that internal field types may have changed since Hibernate Search 5; see Data format and schema changes.

`org.apache.lucene.search.Sort` / `SortField` ⇒ `SearchSort`

Lucene sort fields are replaced with Lucene-independent "search sorts" in Hibernate Search 6.

Most of the time, code that builds queries does not need to manipulate search sorts directly, thanks to the lambda syntax. However, it's still possible to manipulate `SearchSort` objects if you need to pass them around from a method to another.

You can find more information about building sorts and details about all available sorts in the dedicated section of the documentation, and instructions to migrate from the Hibernate Search 5 Query DSL in `Sort` DSL migration reference.

As to adding sorts to search queries, let's consider the query below:

```
MySearchParameters params = ...;
FullTextEntityManager fullTextEntityManager = Search.getFullTextEntityManager( em );
QueryBuilder qb = fullTextEntityManager.getSearchFactory()
        .buildQueryBuilder().forEntity( Book.class ).get();

FullTextQuery fullTextQuery = fullTextEntityManager.createFullTextQuery( qb.all().createQuery
(), Book.class );
fullTextQuery.setFirstResult( params.getPageIndex() * params.getPageSize() );
fullTextQuery.setMaxResults( params.getPageSize() );

fullTextQuery.sort( qb.sort().byField( "pageCount" ).desc()
        .andByField( "title_sort" )
        .createSort() );

List hits = fullTextQuery.getResultList();
```

It would look like this in Hibernate Search 6:

```
MySearchParameters params = ...;
SearchSession session = Search.session( entityManager );
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .sort( f -> f.field( "pageCount" ).desc()
                .then().field( "title_sort" ) )
        .fetchHits( params.getPageIndex() * params.getPageSize(), params.getPageSize() );
```

`Sort` DSL migration reference

The code below makes some assumptions:

- For Hibernate Search 5, a `QueryBuilder` was retrieved from the `SearchFactory` and put in variable `qb`.
- For Hibernate Search 6, the sort is being built in a lambda expression: `Search.session( entityManager ).search( Book.class ).where( f → f.matchAll() ).sort( f → … )`.

Hibernate Search 5	Hibernate Search 6	Documentation
`qb.sort().byScore()` `.createSort()`	`f.score()`	`score`

Hibernate Search 5	Hibernate Search 6	Documentation
`qb.sort().byIndexOrder()` `.createSort()`	`f.indexOrder()`	`indexOrder`
`qb.sort().byField("field")` `.createSort()`	`f.field("field")`	`field` Missing values are sorted differently by default. See When sorting by field value or distance, missing values are last by default.

`qb.sort().byField("field")`
`.asc()`
`.createSort()`
	`f.field("field").asc()`	

`qb.sort().byField("field")`
`.desc()`
`.createSort()`
	`f.field("field").desc()`	

`qb.sort().byField("field")`
`.onMissingValue()`
`.sortFirst()`
`.createSort()`
	`f.field("field")`
	`.missing().first()`	

`qb.sort().byField("field")`
`.onMissingValue()`
`.sortLast()`
`.createSort()`
	`f.field("field")`
	`.missing().last()`	

`qb.sort().byField("field")`
`.onMissingValue()`
`.use("value")`
`.createSort()`
	`f.field("field")`
	`.missing().use("value")`	

`qb.sort()`
`.byField("field",`
 	
`SortField.Type.STRING)`
`.createSort()`
	-	This method was deprecated in Hibernate Search 5.
		
		There is no equivalent in Hibernate Search 6.

`qb.sort().byDistance().onField(`
`"field")`
`.fromLatitude(latitude)`
 	
`.andLongitude(longitude)`
`.createSort()`
	`f.distance("field",`
	`GeoPoint.of(latitude,`
	`longitude))`
		`distance`
		
		Missing values are sorted differently by default. See When sorting by field value or distance, missing values are last by default.

Hibernate Search 5	Hibernate Search 6	Documentation

`qb.sort().byNative(sortField)`	`f.extension(LuceneExtension.get())`
	
	`.fromLuceneSortField(sortField)`
		`fromLuceneSortField`
		
		Be aware that internal field types may have changed since Hibernate Search 5; see Data format and schema changes.
		
		In particular, Hibernate Search 6 relies on `SORTED_SET` docvalues for most field types, so the classic `SortField.TYPE.STRING` and similar just won't work: you need to go through the DSL to create the appropriate sorts.

`qb.sort().byNative("authors.name", "{'order':'asc', 'mode': 'min'}")`	`f.extension(ElasticsearchExtension.get())`
	
	`.fromJson("{'authors.name': {'order':'asc', 'mode': 'min'} }")`
		`fromJson`
		
		Be aware that internal field types may have changed since Hibernate Search 5; see Data format and schema changes.

Native sorts

It is still possible to rely on native Lucene sort fields (e.g. `new SortField(…)`) or Elasticsearch sorts (e.g. `{'title_sort': {…}}`) in Hibernate Search 6: you will just need to rely on the backend-specific extension.

Projections

Basics

Projections gain a full-blown DSL in Hibernate Search 6, allowing more complex projections, as explained in the dedicated section of the documentation.

The `ProjectionConstants` are gone, and the projection DSL must be used instead to build `SearchProjection` objects. You will find instructions to migrate from `ProjectionConstants` to the Hibernate Search 6 projection DSL in `ProjectionConstants/ElasticsearchProjectionConstants` migration reference.

As to adding projections to search queries, let's consider the query below:

```
FullTextEntityManager fullTextEntityManager = Search.getFullTextEntityManager( em );
QueryBuilder qb = fullTextEntityManager.getSearchFactory()
        .buildQueryBuilder().forEntity( Book.class ).get();

FullTextQuery fullTextQuery = fullTextEntityManager.createFullTextQuery( qb.all().createQuery
(), Book.class );
fullTextQuery.setMaxResults( 20 );

fullTextQuery.setProjection( "isbn", "title" );

List hits = fullTextQuery.getResultList();
```

It would look like this in Hibernate Search 6:

```
SearchSession session = Search.session( entityManager );
List<List<?>> hits = searchSession.search( Book.class )
        .select( f -> f.composite(
                f.field( "isbn" ),
                f.field( "title" )
        ) )
        .where( f -> f.matchAll() )
        .fetchHits( 20 );
```

Alternatively, the composite projection can be made more type-safe:

```
SearchSession session = Search.session( entityManager );
List<Map.Entry<ISBN, String>> hits = searchSession.search( Book.class )
        .select( f -> f.composite(
                Map::entry,
                f.field( "isbn", ISBN.class ),
                f.field( "title", String.class )
        ) )
        .where( f -> f.matchAll() )
        .fetchHits( 20 );
```

`ProjectionConstants`/`ElasticsearchProjectionConstants` migration reference

The code below makes some assumptions for Hibernate Search 6: the projection is being built in a lambda expression, e.g. `Search.session( entityManager ).search( Book.class ).select( f → … )`.

Hibernate Search 5 `ProjectionConstants` or `ElasticsearchProjectionConstants`	Hibernate Search 6	Documentation
`THIS`	`f.entity()`	`entity`
`DOCUMENT`	`f.extension(LuceneExtension.get())` `.document()`	`document`
`SCORE`	`f.score()`	`score`

Hibernate Search 5 `ProjectionConstants` or `ElasticsearchProjectionConstants`	Hibernate Search 6	Documentation
`ID`	`f.composite(EntityReference::id, f.entityReference())`	`entityReference`, `composite` Alternatively, `f.entityReference()` (without the wrapping in the "composite" projection) will return an instance of `EntityReference`, which includes both the type and ID of the entity.
`DOCUMENT_ID`	`f.composite(DocumentReference::id, f.documentReference())`	`documentReference`, `composite` This no longer returns the internal Lucene document ID (which can change from one query execution to the next), but instead returns the Hibernate Search document ID, i.e. the String version of the property annotated with `@Id` or `@DocumentId`. Note that you no longer need the internal Lucene document ID to get an explanation of the score. Alternatively, `f.documentReference()` (without the wrapping in the "composite" projection) will return an instance of `DocumentReference`, which includes both the type and ID of the document.

Hibernate Search 5 `ProjectionConstants` or `ElasticsearchProjectionConstants`	Hibernate Search 6	Documentation

`EXPLANATION`	`f.extension(LuceneExtension.get())`
	`.explanation()`
	
	OR
	
	`f.extension(ElasticsearchExtension.get())`
	`.explanation()`
		`explanation` (Lucene)
		
		`explanation` (Elasticsearch)
		
		This projection returns a `JsonObject` for Elasticsearch.

`OBJECT_CLASS`	`f.composite(EntityReference::type,` `f.entityReference())`	`entityReference`, `composite` Alternatively, `f.entityReference()` (without the wrapping in the "composite" projection) will return an instance of `EntityReference`, which includes both the type and ID of the entity.
`SPATIAL_DISTANCE`	`f.distance("field",` `GeoPoint.of(latitude, longitude))`	`distance`

	`f.distance("field",`
	`GeoPoint.of(latitude, longitude))`
	`.unit(`
	`DistanceUnit.KILOMETERS )`	

`SOURCE`	`f.extension(ElasticsearchExtension.get())` `.source()`	`source` This projection used to return a `String` in Hibernate Search 5, but returns a `JsonObject` in Hibernate Search 6.
`TOOK`	–	See `took` and `timedOut`
`TIMED_OUT`	–	See `took` and `timedOut`

Facets ⇒ Aggregations

Facets are now called aggregations, which are a generalization of the concept of faceting.

Like other concepts (predicates, sorts, ...) aggregations have a dedicated DSL in Hibernate Search 6, as explained in the dedicated section of the documentation.

See the following sections for the equivalent aggregation for each type of facet.

One difference with Hibernate Search 5 is that Hibernate Search 6 aggregations no longer allow drill-down (`.selectFacets(…)`). See Drill-down with `.selectFacets`.

Discrete faceting

Let's consider the query below:

```
FullTextEntityManager fullTextEntityManager = Search.getFullTextEntityManager( em );
QueryBuilder qb = fullTextEntityManager.getSearchFactory()
        .buildQueryBuilder().forEntity( Book.class ).get();

FacetingRequest genreFacetingRequest = qb.facet()
        .name( "genreFaceting" )
        .onField( "genre" )
        .discrete()
        .orderedBy( FacetSortOrder.COUNT_DESC )
        .includeZeroCounts( false )
        .maxFacetCount( 3 )
        .createFacetingRequest();

FullTextQuery fullTextQuery = fullTextEntityManager.createFullTextQuery( qb.all().createQuery
(), Book.class );
fullTextQuery.setMaxResults( 20 );

FacetManager facetManager = fullTextQuery.getFacetManager();
facetManager.enableFaceting( genreFacetingRequest );

List hits = fullTextQuery.getResultList();

List<Facet> facets = facetManager.getFacets( "genreFaceting" );
```

It would look like this in Hibernate Search 6:

```
SearchSession session = Search.session( entityManager );

AggregationKey<Map<Genre, Long>> countByGenreKey = AggregationKey.of( "countByGenre" );

SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .aggregation( countByGenreKey, f -> f.terms()
                .field( "genre", Genre.class )
                .orderByCountDescending()
                .minDocumentCount( 1 )
                .maxTermCount( 3 ) )
        .fetch( 20 );

List<Book> hits = result.hits();

Map<Genre, Long> countByGenre = result.aggregation( countByGenreKey );
```

See this section of the documentation for more information.

Range faceting

Let's consider the query below:

```
FullTextEntityManager fullTextEntityManager = Search.getFullTextEntityManager( em );
QueryBuilder qb = fullTextEntityManager.getSearchFactory()
        .buildQueryBuilder().forEntity( Book.class ).get();

FacetingRequest priceFacetingRequest = qb.facet()
        .name( "priceFaceting" )
        .onField( "price" )
        .range()
        .below( 1000 ).excludeLimit()
        .from( 1001 ).to( 1500 ).excludeLimit()
        .above( 1500 )
        .orderedBy( FacetSortOrder.COUNT_DESC )
        .includeZeroCounts( false )
        .maxFacetCount( 3 )
        .createFacetingRequest();

FullTextQuery fullTextQuery = fullTextEntityManager.createFullTextQuery( qb.all().createQuery
(), Book.class );
fullTextQuery.setMaxResults( 20 );

FacetManager facetManager = fullTextQuery.getFacetManager();
facetManager.enableFaceting( priceFacetingRequest );

List hits = fullTextQuery.getResultList();

List<Facet> facets = facetManager.getFacets( "priceFaceting" );
```

It would look like this in Hibernate Search 6:

```
SearchSession session = Search.session( entityManager );

AggregationKey<Map<Range<Double>, Long>> countByPriceRangeKey = AggregationKey.of(
"countByPriceRange" );

SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .aggregation( countByPriceRangeKey, f -> f.range()
                .field( "price", Double.class )
                .range( Range.lessThan( 1000.0 ) )
                .range( Range.canonical( 1000.0, 1500.0 ) )
                .range( Range.atLeast( 1500.0 ) )
                // Not equivalent to 'orderedBy'
                // Not equivalent to 'includeZeroCounts'
                // Not equivalent to 'maxFacetCount'
        .fetch( 20 );

List<Book> hits = result.hits();

Map<Range<Double>, Long> countByPriceRange = result.aggregation( countByPriceRangeKey );
```

See this section of the documentation for more information.

`orderedBy`, `includeZeroCounts` and `maxFacetCount` have no equivalent Hibernate Search 6 range aggregations: all given ranges will always be included in the resulting `Map`.

The behavior of these methods can be implemented by post-processing the `Map` in user code.

Drill-down with `.selectFacets`

In Hibernate Search 5, the `.selectFacets` method used to allow "drill-down", i.e. adding a filter to the query to only consider documents in a given facet.

Hibernate Search 6 no longer supports this feature directly.

To perform a drill-down, create a new query with the original predicate wrapped in a boolean predicate and add a `filter` clause to restrict the hits to the selected facet(s):

- For discrete faceting, use a `match` predicate.
- For range faceting, use a `range` predicate.

Error handler

The `ErrorHandler` interface was replaced with the `FailureHandler` interface, and the related configuration properties changed.

See this section of the documentation for more information about background failure handling.

Also, be aware that the `MassIndexer` now exposes a `failureHandler` parameter, to handle failures during mass indexing differently (e.g. report to the web console from which mass indexing was initiated). More information here.

Full-text filter

Hibernate Search 6.0 does not support named full-text filters at the moment. In most cases, you can replace them with static methods.

For example, let's take this filter from a Hibernate Search 5 application:

```
public class SecurityFilterFactory {
    private Integer level;

    /**
     * injected parameter
     */
    public void setLevel(Integer level) {
        this.level = level;
    }

    @Factory
    public Query getFilter() {
        return new TermQuery( new Term( "level", level.toString() ) );
    }
}
```

```
@Entity
@Indexed
@FullTextFilterDef(name = "security", impl = SecurityFilterFactory.class)
public class Driver {
   // ...
}
```

The filter could be used this way:

```
FullTextEntityManager fullTextEntityManager = Search.getFullTextEntityManager( em );
QueryBuilder qb = fullTextEntityManager.getSearchFactory()
        .buildQueryBuilder().forEntity( Driver.class ).get();

BooleanJunction junction = qb.bool();
junction.must( qb.all().createQuery() );

// Not shown: add user predicates to the junction (search terms, etc.).
// ...

org.apache.lucene.search.Query luceneQuery = junction.createQuery();

FullTextQuery fullTextQuery = fullTextEntityManager.createFullTextQuery( luceneQuery, Driver
.class );
fullTextQuery.setMaxResults( 20 );

// HERE: Enable the filter
fullTextQuery.enableFullTextFilter("security").setParameter("level", 5);

List hits = fullTextQuery.getResultList();
```

In Hibernate Search 6, you can define the filter as a static method:

```
public final class SecurityFilterFactory {
    private SecurityFilterFactory() {
    }

    public static SearchPredicate create(SearchPredicateFactory factory, int level) {
        return factory.match().field( "level" ).matching( level.toString() );
    }
}
```

And then simply add it to the root boolean predicate when searching:

```
SearchSession session = Search.session( entityManager );
List<Book> hits = searchSession.search( Driver.class )
        .where( f -> f.bool( b -> {
            b.must( f.matchAll() );
            // HERE: Enable the filter
            b.filter( SecurityFilterFactory.create( f, 5 ) );

            // Not shown: add user predicates to the junction (search terms, etc.).
            // ...
        } ) )
        .fetchHits( 20 );
```

If you have needs that can be addressed only with named full-text filters, and cannot be solved with the solution above, consider upgrading directly to Hibernate Search 6.1, which provides a similar feature called named predicates.

`@Factory`

The `@Factory` annotation does not exist in Hibernate Search 6 anymore.

You are encouraged to rely on a proper dependency injection framework if you need such a feature: just reference the bean name instead of referencing the bean class in your Hibernate Search mapping/configuration. See the section of the documentation about beans in Hibernate Search for details and supported DI frameworks.

If you don't use a dependency injection framework, here are details on how to migrate:

String bridges, field bridges, class bridges

Use their `*Binder` equivalent in Hibernate Search 6, which can act as a factory: `ValueBinder`, `PropertyBinder`, `TypeBinder`.

Full-text filters

These no longer exist in Hibernate Search 6. See Full-text filter.

Programmatic mapping

`@Factory` is no longer needed for the programmatic mapping, since you will pass a callback (`HibernateOrmSearchMappingConfigurer`) instead of passing the mapping directly. Whatever code was implemented in your factory can be moved to the configurer.

Analysis definition providers

Analysis definition providers are now called analysis configurers, and as they are just callbacks that are used only once, the `@Factory` annotation should not be necessary. Whatever code was implemented in your factory can be moved to the configurer.

`SearchException`

`org.hibernate.search.exception.SearchException` has moved to `org.hibernate.search.util.common.SearchException`.

Sharding: `IndexShardingStrategy`/`ShardIdentifierProvider`/`ShardIdentifierProviderTemplate`

Static sharding is still available in Hibernate Search 6, but it works differently, so the Hibernate Search 5 APIs are no longer available. To implement static sharding in Hibernate Search 6, refer to this section of the documentation.

Dynamic sharding is no longer available in Hibernate Search 6. If your application absolutely requires it, contact us so we can discuss how to best address your needs, or drop a comment here.

`SearchFactory`

Basics

The equivalent to Hibernate Search 5's `SearchFactory` is Hibernate Search 6's `SearchMapping`, but some operations are more conveniently accessible directly from `SearchSession`.

Here is how to retrieve the `SessionFactory` in Hibernate Search 5:

```
SearchFactory searchFactory = Search.getFullTextEntityManager( entityManager )
.getSearchFactory();
// OR
SearchFactory searchFactory = Search.getFullTextSession( session ).getSearchFactory();
```

The main entry point to Hibernate Search APIs is still named `Search`, but it moved to another package: `org.hibernate.search.mapper.orm.Search`. Here is how to retrieve the `SearchMapping` in Hibernate Search 6:

```
SearchMapping mapping = Search.mapping( entityManagerFactory );
// OR
SearchMapping mapping = Search.mapping( sessionFactory );
// OR
SearchMapping mapping = Search.mapping( entityManager.getEntityManagerFactory() );
// OR
SearchMapping mapping = Search.mapping( session.getSessionFactory() );
```

See the following sections for the exact equivalent of each operation of `SearchFactory`.

Merging segments: `SearchFactory.optimize()`/`SearchFactory.optimize(…)`

In Hibernate Search 5:

```
SearchFactory searchFactory = Search.getFullTextSession( session ).getSearchFactory();

searchFactory.optimize( Book.class );
// OR
searchFactory.optimize();
```

Equivalent in Hibernate Search 6:

```
SearchSession searchSession = Search.session( session );

searchSession.workspace( Book.class ).mergeSegments();
// OR
searchSession.workspace().mergeSegments();
```

See this section of the documentation for more information.

Getting Lucene analyzers: `SearchFactory.getAnalyzer(…)`

In Hibernate Search 5:

```
SearchFactory searchFactory = Search.getFullTextSession( session ).getSearchFactory();

// Get a named analyzer
Analyzer analyzer = searchFactory.getAnalyzer( "customanalyzer" );

// Get the search analyzer for a given indexed type
Analyzer searchAnalyzer = searchFactory.getAnalyzer( Book.class );
```

Equivalent in Hibernate Search 6:

```
SearchMapping mapping = Search.mapping( entityManagerFactory );
// OR
SearchMapping mapping = Search.mapping( entityManager.getEntityManagerFactory() );

// Get a named analyzer
Optional<? extends Analyzer> analyzer = mapping.backend().unwrap( LuceneBackend.class )
        .analyzer( "customanalyzer" );

// Get the search analyzer for a given indexed type
Analyzer searchAnalyzer = mapping.indexedEntity( Book.class ).indexManager()
        .unwrap( LuceneIndexManager.class ).searchAnalyzer();

// Get the search analyzer for a given index
Analyzer searchAnalyzer = mapping.indexManager( "Book" ).indexManager()
        .unwrap( LuceneIndexManager.class ).searchAnalyzer();
```

See this section of the documentation for more information.

Building queries: `SearchFactory.buildQueryBuilder()`

See `org.apache.lucene.search.Query` ⇒ `SearchPredicate`.

Statistics: `SearchFactory.getStatistics()`

Hibernate Search 6 does not provide statistics at the moment.

The current plans are to implement support for tracing in a future release (HSEARCH-4057). This would provide a more powerful solution to users looking for insight into the behavior of their application.

If you need this feature urgently, we'll gladly help anyone interested in contributing a patch: feel free to contact us.

Accessing Lucene index readers: `SearchFactory.getIndexReaderAccessor`

Hibernate Search 6.0 does not provide direct access to the index reader.

If you need direct access to an index reader, consider upgrading directly to Hibernate Search 6.1, which provides just that feature.

Metamodel: `SearchFactory.getIndexedTypeDescriptor`, `SearchFactory.getIndexedTypes`

The metamodel is still available in Hibernate Search 6, but through different API calls. See this section of the documentation for more information.

In Hibernate Search 6, the metamodel includes:

- General information about indexed types (entity type, JPA entity name)
- Detailed information about the indexes and all their declared fields.

Compared to Hibernate Search 5, it **no longer** includes:

- Detailed information about indexed types and their annotated properties.

Accessing the Elasticsearch client: `SearchFactory.getIndexFamily`

The main purpose of `getIndexFamily` in Hibernate Search 5 was to access the Elasticsearch REST client.

See this section of the documentation for an equivalent solution in Hibernate Search 6.

`MassIndexer`

Basics

The `MassIndexer` mostly stays the same in Hibernate Search 6, but it moved to a different package: `org.hibernate.search.mapper.orm.massindexing.MassIndexer`.

Here is how to retrieve and use a `MassIndexer` in Hibernate Search 5:

```
FullTextSession fullTextSession = Search.getFullTextSession( session );
fullTextSession.createIndexer().startAndWait();
// OR
fullTextSession.createIndexer( Book.class, Author.class ).startAndWait();
```

Here is how to retrieve and use a `MassIndexer` in Hibernate Search 6:

```
SearchSession searchSession = Search.session( entityManager );
searchSession.massIndexer().startAndWait();
// OR
searchSession.massIndexer( Book.class, Author.class ).startAndWait();
```

Most mass indexing parameters stayed the same; see the following sections for the exact equivalent of parameters that changed.

See this section of the documentation for more information about the mass indexer.

`MassIndexer.optimizeOnFinish(…)`

The equivalent method in Hibernate Search 6 is `MassIndexer.mergeSegmentsOnFinish(…)`.

`MassIndexer.optimizeAfterPurge(…)`

The equivalent method in Hibernate Search 6 is `MassIndexer.mergeSegmentsAfterPurge(…)`.

`MassIndexer.progressMonitor(…)`

The equivalent method in Hibernate Search 6 is `MassIndexer.monitor(…)`. Instead of expecting an instance of `MassIndexerProgressMonitor`, it expects an instance of `MassIndexingMonitor`, which is largely the same except that all methods take `long` arguments instead of `int`.

`MassIndexer.threadsForSubsequentFetching(…)`

This method was deprecated in Hibernate Search 5 and didn't do anything.

It is no longer available in Hibernate Search 6.

Batch (JSR-352) integration

The mass indexing Batch (JSR-352) job mostly stays the same in Hibernate Search 6, but `MassIndexingJob` moved to a different package: `org.hibernate.search.batch.jsr352.core.massindexing.MassIndexingJob`.

See this section of the documentation for more information about the JSR-352 integration.

See the table below for the exact equivalent of parameters that changed.

Table 1. Job Parameters in JSR 352 Integration that changed in Hibernate Search 6

Hibernate Search 5	Hibernate Search 6	Comment
`optimizeAfterPurge`, `optimizeAfterPurge(boolean)`	`mergeSegmentsAfterPurge`, `mergeSegmentsAfterPurge(boolean)`	
`optimizeOnFinish`, `optimizeOnFinish(boolean)`	`mergeSegmentsOnFinish`, `mergeSegmentsOnFinish(boolean)`	
`restrictedBy(Criterion)` (`MassIndexingJob` parameter builder only)	`customQueryHQL`, `restrictedBy(String)`	Using Hibernate ORM criteria is no longer supported. Use HQL / JPQL instead. See this section of the documentation for more details and limitations.

SPI changes

Due to the extensive rewrites involved in Hibernate Search 6, existing integrations relying on Hibernate Search 5 are likely to require a full rewrite.

We will be glad to help, so feel free to contact us.

Behavior changes

No default bridge for `java.util.Class`

There is no longer a builtin, default bridge for `java.util.Class`.

If you need to index a `Class<?>`, you will need to implement a custom bridge (probably from `Class` to `String`).

Optionally, you can also register your custom bridge as a default bridge so that it is applied automatically and transparently to all fields defined on properties of type `Class`.

The document ID is not an index field

Hibernate Search 5 used to create an index field for document IDs, which could then be queried using the query DSL in particular:

```
@Indexed
@Entity
public class MyEntity {

    @Id // Implicitly creates a String field named "id"
    public Integer id;

}
```

In Hibernate Search 6, this is no longer the case. The identifier "field" is internal, and is not named after the entity property it originates from.

If you want to use the identifier as a field:

- For exact matches, consider the new `id` predicate
- For any other reason (sorts, aggregations, non-exact matches, ...), declare a field explicitly by annotating the identifier property with `@GenericField` or a similar annotation.

`@Indexed` is inherited

In Hibernate Search 5, `@Indexed` was not inherited by subclasses; the following configuration would result in instances of `MyEntity` being indexed, but not instances of `MySubClassEntity`:

```
@Indexed
@Entity
public class MyEntity {
    // ...
}

@Entity
public class MySubClassEntity extends MyEntity {
    // ...
}
```

In Hibernate Search 6, `@Indexed` **is** inherited by subclasses; the configuration above would result in instances of both `MyEntity` and `MySubClassEntity` being indexed, though in different indexes. A search query on type `MyEntity` will return instances of both `MyEntity` and `MySubClassEntity`, in accordance with the Liskov substitution principle.

To prevent `MySubClassEntity` from being indexed, annotate it with `@Indexed(enabled = false)`; this will restore the Hibernate Search 5 behavior:

```
@Indexed
@Entity
public class MyEntity {
    // ...
}

@Indexed(enabled = false)
@Entity
public class MySubClassEntity extends MyEntity {
    // ...
}
```

One index ⇒ one type

In Hibernate Search 5, it used to be possible to store multiple entity types in the same index.

In Hibernate Search 6, each type must have its own dedicated index. Annotating two distinct types with `@Indexed(index = "my-index")` (the same index name) will lead to a bootstrap failure.

The index name defaults to the entity name

In Hibernate Search 5, the default name of the index for a given entity when `@Indexed.index` was not set was the fully-qualified class name, so the following mapping resulted in an index named `com.mycompany.MyEntity1` and another named `com.mycompany.MyEntity2`:

```
package com.mycompany;

@Indexed
@Entity
public class MyEntity1 {
    // ...
}

@Indexed
@Entity(name = "MyEntityTwo")
public class MyEntity2 {
    // ...
}
```

In Hibernate Search 6, the default name of the index in that case is the JPA entity name, so the mapping above results in an index named simply `MyEntity1` and another named `MyEntityTwo`.

`indexNullAs` is irrelevant when building predicates and when projecting

In Hibernate Search 5, when a field was configured with `indexNullAs`:

- passing `null` as the value to match for a query created through the Search DSL would automatically translate to the null token.
- projecting on a field whose value is equal to the configured null token would return `null`.

In Hibernate Search 6, this is no longer the case:

- passing `null` as the value to match for a predicate created through the Search DSL will throw an exception (since `null` is not indexed, only the corresponding null token is).
- projecting on a field whose value is equal to the configured null token will return that null token (since Hibernate Search can't tell whether the indexed value was `null` or just happened to be equal to the null token).

If this behavior is not acceptable, consider dropping the `indexNullAs` option and relying on the `exists` predicate to search for documents that have or don't have a value for a given field.

Faceting returns normalized strings

In Hibernate Search 5, a terms aggregation on a field with a normalizer used to return raw (non-normalized) values.

In Hibernate Search 6, the same aggregation on this normalized field will return normalized values.

`@IndexedEmbedded` on an association enables reindexing when the target entity changes by default

In Hibernate Search 5, automatic reindexing was only triggered by changes to the indexed entity by default: if you were to add `@IndexedEmbedded(includePaths = "name")` to an association from entity `A` to another entity `B`, then calling `a.setB(…)` would lead to reindexing of `a`, but calling `a.getB().setName("a new name")` would not. In order to make sure that changes to `B.name` automatically trigger reindexing of the associated `A`, you had to mark the inverse side of the association, `B.a`, with the `@ContainedIn` annotation.

This behavior changed in Hibernate Search 6: in the example above, Hibernate Search 6 would automatically trigger reindexing of `B.a` when `B.name` is modified, **without** the need of applying a `@ContainedIn` annotation.

This means that automatic indexing is "safe" by default: whenever you apply `@IndexedEmbedded` to an association, Hibernate Search will automatically resolve the inverse side of that association and will make sure that any change in "indexed-embedded" entities lead to reindexing of affected indexed entities.

This also means that Hibernate Search 6 needs to know the inverse side of associations annotated with `@IndexedEmbedded`. If it can't resolve it (for example when an `@IndexedEmbedded` association has no inverse side), Hibernate Search 6 will raise errors at bootstrap, looking like this:

```
Unable to find the inverse side of the association on type 'A' at path '.b<no value
extractors>'
Hibernate Search needs this information in order to reindex 'A' when 'B' is modified.
You can solve this error by defining the inverse side of this association,
either with annotations specific to your integration
(@OneToMany(mappedBy = ...) in Hibernate ORM)
or with the Hibernate Search @AssociationInverseSide annotation.
Alternatively, if you do not need to reindex 'A' when 'B' is modified,
you can disable automatic reindexing with
@IndexingDependency(reindexOnUpdate = ReindexOnUpdate.SHALLOW).
```

The error message already explains the potential solutions, but here is a perhaps clearer explanation of each solution in the context of migrating Hibernate Search 5 applications:

1. Add an inverse side to the assocation in `B`: `@OneToMany(mappedBy = …) List<A> a` or `@OneToOne(mappedBy = …) A a` or `@ManyToMany(mappedBy = …) List<A> a`. This is obviously only practical if `B.a` has a reasonably low cardinality (a few dozens instances of `A` for a given `B`).

   

   Make sure to always update both sides of the association when you change it!

   In some rare situations, the inverse side of the association cannot be expressed with `mappedBy`; in this case, you can use `@AssociationInverseSide`, an annotation specific to Hibernate Search.

2. Annotate the `@IndexedEmbedded` association with `@IndexingDependency(reindexOnUpdate = ReindexOnUpdate.SHALLOW)`, which will restore the behavior of Hibernate Search 5 when `@ContainedIn` was not used. This is particularly useful when the association is massive and highly asymmetric (e.g. hundreds or more instances of `A` for just a dozen instances of `B`) and `B` is not expected to change (reference data, e.g. a `Country` entity).

For more information, refer to Tuning automatic reindexing, in particular this section for shallow reindexing and this section for specifying the inverse side of an association.

Indexed, `@Transient` properties require additional configuration

In Hibernate Search 5, when a `@Transient` property was mapped to an index field, Hibernate Search would switch to a "non-optimized" mode where reindexing was triggered for any change in the same entity, assuming that the transient property value was computed from other properties of the same entity. This would work fine in some cases, but could lead to unnecessary reindexing when unrelated properties were modified and to out-of-sync indexes when the transient property was computed from data extracted from associated entities.

This behavior changed in Hibernate Search 6: if a `@Transient` property is mapped to an index field, Hibernate Search will trigger a bootstrap failure asking for additionnal configuration.

The "right" solution in that case is to put an annotation on the transient property, to explain which properties are used to derive the value of the transient property: `@IndexingDependency(derivedFrom = …)`. You can find more information in this section of the documentation.

Alternatively, if you do not care about automatic reindexing when the transient property changes, then you can just disable automatic reindexing for this property with `@IndexingDependency(reindexOnUpdate = ReindexOnUpdate.NO)`. See this section of the documentation.

Container extraction for `@*Field` is implied

In Hibernate Search 6, when a `@*Field` annotation is applied on a property of a container type, the field will work on container **elements** by default, not on the container itself.

For example, the following mapping will create a `String` field, and Hibernate Search will index each element of the `List` in the `notes` index field:

```
@FullTextField
@ElementCollection
private List<String> notes;
```

This should be kept in mind when migrating custom bridges that work on container types. For example, the following mapping likely won't work, because `MyListBridge` expects values of type `List`, not `String`:

```
@GenericField(valueBridge = @ValueBridgeRef(type = MyListBridge.class)) // Will not work
@ElementCollection
private List<String> notes;
```

To disable the automatic extraction of container element (and get back to the Hibernate Search 5 behavior), use `extraction = @ContainerExtraction(extract = ContainerExtract.NO)`:

```
@GenericField(valueBridge = @ValueBridgeRef(type = MyListBridge.class),
        extraction = @ContainerExtraction(extract = ContainerExtract.NO)) // Will work
@ElementCollection
private List<String> notes;
```

See this section of the documentation for more information.

More fields are numeric by default

Fields of type `Boolean`, `boolean`, `Short`, `short`, `Byte` and `byte` used to be indexed as string values by default in Hibernate Search 5, but are indexed as numeric values by default in Hibernate Search 6.

Similarly, fields created to represent the embedded identifier when using `@IndexedEmbedded.includeEmbeddedObjectId` used to be string fields, unless annotated with `@NumericField`. The behavior changed in Hibernate Search 6:

- If the embedded entity's document identifier has a custom identifier bridge (`@DocumentId(identifierBridge = …)`), then the field representing the embedded identifier will be a string field.
- Otherwise, the field representing the embedded identifier will be assigned a type by the default value bridge for its type. This means in particular that a `Long` embedded identifier will be represented by a `Long` embedded field, an `Instant` embedded identifier will be represented by an `Instant` embedded field, etc.

Scrolling is forward-only

When using a `Query` that is an adapter to a Hibernate Search query, the `ScrollableResults` returned by `Query.scroll()`/`Query.scroll(ScrollMode)` is now always forward-only; trying to move the cursor backward will lead to an exception being thrown.

Scrolling/`Query.iterate()` load entities eagerly in batches

When using a `Query` that is an adapter to a Hibernate Search query, the `ScrollableResults` returned by `Query.scroll()`/`Query.scroll(ScrollMode)`, as well as the `iterate()` method, no longer load entities lazily as late as possible: instead, entities are loaded in batches under the hood.

This is important in particular if you clear the session, as you might clear an entity that was already loaded as part of the current batch, which could result in a `LazyInitializationException` being thrown upon accessing that entity.

You can control the size of batches by configuring the loading fetch size or scrolling through Hibernate Search's own APIs.

When sorting by field value or distance, missing values are last by default

In Hibernate Search 5, sorts involving missing values used to assume a default value depending on the field type, e.g. `0` for numeric types. As a result, a document whose field does not have a value could end up first, or right in the middle of other documents after a sort.

In Hibernate Search 6, a document whose field does not have a value always ends up at the very end by default. This behavior can be customized when defining the sort.

Logger `org.hibernate.search.query` uses the `TRACE` level

Hibernate Search logs all executed search queries to logger `org.hibernate.search.query`.

In Hibernate Search 5, these logs were at the `DEBUG` level. In Hibernate Search 5, they are at the `TRACE` level.

The default similarity for Lucene is now BM25

The default similarity for the Lucene backend is now `BM25Similarity`, instead of `ClassicSimilarity` in Hibernate Search 5.

If necessary, you can force the `ClassicSimilarity` through an analysis definition provider.