Hibernate Search 8.4.0.CR1

Reference Documentation

2026-05-11

Table of Contents

Preface . . . 1
1. Compatibility . . . 2
1.1. Dependencies . . . 2
1.1.1. Adding Hibernate Search dependencies to your project . . . 2
Hibernate Search BOM . . . 2
Hibernate Search Platform . . . 3
1.2. Framework support . . . 3
1.2.1. Quarkus . . . 3
1.2.2. WildFly . . . 4
1.2.3. Spring Boot . . . 4
Configuration properties . . . 4
Dependency versions . . . 4
Application hanging on startup . . . 5
Spring Boot's Elasticsearch client and auto-configuration . . . 5
1.2.4. Other . . . 6
2. Getting started with Hibernate Search . . . 7
3. Migrating . . . 8
4. Concepts . . . 9
4.1. Full-text search . . . 9
4.2. Entity types . . . 9
4.3. Mapping . . . 9
4.4. Binding . . . 10
4.5. Analysis . . . 10
4.6. Commit and refresh . . . 11
4.7. Sharding and routing . . . 11
5. Architecture . . . 13
5.1. Components of Hibernate Search . . . 13
5.2. Examples of architectures . . . 14
5.2.1. Overview . . . 14
5.2.2. Single-node application with the Lucene backend . . . 14
Description . . . 14
Pros and cons . . . 15
Getting started . . . 15
5.2.3. Single-node or multi-node application, without coordination and with the Elasticsearch backend . . . 15
Description . . . 15
Pros and cons . . . 16
Getting started . . . 16
5.2.4. Multi-node application with outbox polling and Elasticsearch backend . . . 17
Description . . . 17
Pros and cons . . . 17
Getting started . . . 18
6. Hibernate ORM integration . . . 19
6.1. Basics . . . 19
6.2. Startup . . . 19
6.3. Shutdown . . . 19
6.4. Mapping `Map`-based models . . . 19
6.5. Multi-tenancy with non-string tenant identifiers . . . 19
6.6. Other configuration . . . 19
7. Standalone POJO Mapper . . . 20
7.1. Basics . . . 20
7.2. Startup . . . 20
7.3. Shutdown . . . 20
7.4. Bean provider . . . 21
7.5. Multi-tenancy . . . 21
7.6. Mapping . . . 22
7.7. Indexing . . . 22
7.7.1. Listener-triggered indexing . . . 22
7.7.2. Explicitly indexing on entity change events . . . 22
7.7.3. Mass indexing . . . 22

7.7.4. Entity loading in search queries . . . 22
7.8. Coordination . . . 23
7.9. Reading configuration properties from a file . . . 23
7.10. Other configuration . . . 23
8. Configuration . . . 24
8.1. Configuration sources . . . 24
8.1.1. Configuration sources when integrating into Hibernate ORM . . . 24
8.1.2. Configuration sources with the Standalone POJO mapper . . . 24
8.2. Configuration properties . . . 24
8.2.1. Structure of configuration properties . . . 24
8.2.2. Building property keys programmatically . . . 25
8.2.3. Type of configuration properties . . . 26
8.3. Configuration property checking . . . 26
8.4. Beans . . . 26
8.4.1. Supported frameworks . . . 26
Supported frameworks when integrating into Hibernate ORM . . . 26
Supported frameworks when using the Standalone POJO Mapper . . . 26
8.4.2. Bean references . . . 27
8.4.3. Parsing of bean references . . . 27
8.4.4. Bean resolution . . . 27
8.4.5. Bean injection . . . 28
8.4.6. Bean lifecycle . . . 28
8.5. Global configuration . . . 28
8.5.1. Background failure handling . . . 28
8.5.2. Multi-tenancy . . . 29
9. Main API Entry Points . . . 31
9.1. `SearchMapping` . . . 31
9.1.1. Basics . . . 31
9.1.2. Retrieving the `SearchMapping` with the Hibernate ORM integration . . . 31
9.1.3. Retrieving the `SearchMapping` with the Standalone POJO Mapper . . . 31
9.2. `SearchSession` . . . 31
9.2.1. Basics . . . 31
9.2.2. Retrieving the `SearchSession` with the Hibernate ORM integration . . . 32
9.2.3. Retrieving the `SearchSession` with the Standalone POJO Mapper . . . 32
9.3. `SearchScope` . . . 33
10. Mapping entities to indexes . . . 34
10.1. Configuring the mapping . . . 34
10.1.1. Annotation-based mapping . . . 34
10.1.2. Classpath scanning . . . 34
Basics . . . 34
Scanning dependencies of the application . . . 34
Configuring scanning . . . 34
10.1.3. Programmatic mapping . . . 35
10.1.4. Mapping configurer . . . 35
Hibernate ORM integration . . . 35
Standalone POJO Mapper . . . 36
10.2. Entity definition . . . 37
10.2.1. Basics . . . 37
10.2.2. Explicit entity definition . . . 37
10.2.3. Entity name . . . 38
10.2.4. Mass loading strategy . . . 38
10.2.5. Selection loading strategy . . . 41
10.2.6. Programmatic mapping . . . 43
10.3. Entity/index mapping . . . 44
10.3.1. Basics . . . 44
10.3.2. Explicit index/backend . . . 44
10.3.3. Conditional indexing and routing . . . 45
10.3.4. Programmatic mapping . . . 45
10.4. Mapping the document identifier . . . 45
10.4.1. Basics . . . 45

10.4.2. Explicit identifier mapping . . . 45
10.4.3. Supported identifier property types . . . 46
10.4.4. Programmatic mapping . . . 48
10.5. Mapping a property to an index field with `@GenericField`, `@FullTextField`, . . . 48
10.5.1. Basics . . . 48
10.5.2. Available field annotations . . . 49
10.5.3. Field annotation attributes . . . 50
10.5.4. Supported property types . . . 55
10.5.5. Support for legacy `java.util` date/time APIs . . . 56
10.5.6. Mapping custom property types . . . 57
10.5.7. Programmatic mapping . . . 57
10.6. Mapping associated elements with `@IndexedEmbedded` . . . 57
10.6.1. Basics . . . 58
10.6.2. `@IndexedEmbedded` and `null` values . . . 61
10.6.3. `@IndexedEmbedded` on container types . . . 61
10.6.4. Setting the object field name with `name` . . . 61
10.6.5. Setting the field name prefix with `prefix` . . . 61
10.6.6. Casting the target of `@IndexedEmbedded` with `targetType` . . . 62
10.6.7. Reindexing when embedded elements change . . . 62
10.6.8. Embedding the entity identifier . . . 62
10.6.9. Filtering embedded fields and breaking `@IndexedEmbedded` cycles . . . 63
10.6.10. Structuring embedded elements as nested documents using `structure` . . . 66
`DEFAULT` or `FLATTENED` structure . . . 66
`NESTED` structure . . . 68
10.6.11. Filtering association elements . . . 69
10.6.12. Programmatic mapping . . . 71
10.7. Mapping container types with container extractors . . . 71
10.7.1. Basics . . . 71
10.7.2. Explicit container extraction . . . 71
10.7.3. Disabling container extraction . . . 72
10.7.4. Programmatic mapping . . . 72
10.8. Mapping geo-point types . . . 73
10.8.1. Basics . . . 73
10.8.2. Using `@GenericField` and the `GeoPoint` interface . . . 73
10.8.3. Using `@GeoPointBinding`, `@Latitude` and `@Longitude` . . . 74
10.8.4. Programmatic mapping . . . 76
10.9. Mapping multiple alternatives . . . 77
10.9.1. Basics . . . 77
10.9.2. Programmatic mapping . . . 79
10.10. Tuning when to trigger reindexing . . . 80
10.10.1. Basics . . . 80
10.10.2. Enriching the entity model with `@AssociationInverseSide` . . . 80
10.10.3. Reindexing when a derived value changes with `@IndexingDependency` . . . 81
10.10.4. Limiting reindexing of containing entities with `@IndexingDependency` . . . 82
`ReindexOnUpdate.SHALLOW`: limiting reindexing to same-entity updates only . . . 82
`ReindexOnUpdate.NO`: disabling reindexing caused by updates of a particular property . . . 84
10.10.5. Programmatic mapping . . . 85
10.11. Changing the mapping of an existing application . . . 86
10.12. Custom mapping annotations . . . 86
10.12.1. Basics . . . 86
10.12.2. Custom root mapping annotations . . . 87
10.13. Inspecting the mapping . . . 87
11. Mapping index content to custom types (projection constructors) . . . 90
11.1. Basics . . . 90
11.2. Detection of mapped projection types . . . 91
11.3. Implicit inner projection inference . . . 91
11.3.1. Basics . . . 91
11.3.2. Inner projection and type . . . 91
11.3.3. Inner projection and field path . . . 92
11.4. Explicit inner projection . . . 92

11.5. Mapping types with multiple constructors . . . 92
11.6. Programmatic mapping . . . 93
12. Binding and bridges . . . 94
12.1. Basics . . . 94
12.2. Value bridge . . . 94
12.2.1. Basics . . . 94
12.2.2. Type resolution . . . 96
12.2.3. Using value bridges in other `@*Field` annotations . . . 96
12.2.4. Supporting projections with `fromIndexedValue()` . . . 96
12.2.5. Parsing the string representation to an index field type with `parse()` . . . 97
12.2.6. Formatting the value as string with `format()` . . . 98
12.2.7. Compatibility across indexes with `isCompatibleWith()` . . . 99
12.2.8. Configuring the bridge more finely with `ValueBinder` . . . 99
12.2.9. Passing parameters . . . 100
Simple, string parameters . . . 100
Parameters with custom annotations . . . 102
12.2.10. Accessing the ORM session or session factory from the bridge . . . 103
12.2.11. Injecting beans into the value bridge or value binder . . . 104
12.2.12. Programmatic mapping . . . 104
12.2.13. Incubating features . . . 105
12.3. Property bridge . . . 105
12.3.1. Basics . . . 105
12.3.2. Passing parameters . . . 107
Simple, string parameters . . . 107
Parameters with custom annotations . . . 108
12.3.3. Accessing the ORM session from the bridge . . . 110
12.3.4. Injecting beans into the binder . . . 110
12.3.5. Programmatic mapping . . . 111
12.3.6. Incubating features . . . 111
12.4. Type bridge . . . 112
12.4.1. Basics . . . 112
12.4.2. Passing parameters . . . 114
Simple, string parameters . . . 114
Parameters with custom annotations . . . 115
12.4.3. Accessing the ORM session from the bridge . . . 117
12.4.4. Injecting beans into the binder . . . 118
12.4.5. Programmatic mapping . . . 118
12.4.6. Incubating features . . . 118
12.5. Identifier bridge . . . 119
12.5.1. Basics . . . 119
12.5.2. Type resolution . . . 120
12.5.3. Compatibility across indexes with `isCompatibleWith()` . . . 120
12.5.4. Parsing identifier's string representation with `parseIdentifierLiteral(..)` . . . 120
12.5.5. Configuring the bridge more finely with `IdentifierBinder` . . . 121
12.5.6. Passing parameters . . . 122
Simple, string parameters . . . 122
Parameters with custom annotations . . . 123
12.5.7. Accessing the ORM session or session factory from the bridge . . . 124
12.5.8. Injecting beans into the bridge or binder . . . 125
12.5.9. Programmatic mapping . . . 125
12.5.10. Incubating features . . . 126
12.6. Routing bridge . . . 126
12.6.1. Basics . . . 126
12.6.2. Using a routing bridge for conditional indexing . . . 126
12.6.3. Using a routing bridge to control routing to index shards . . . 127
12.6.4. Passing parameters . . . 129
12.6.5. Accessing the ORM session from the bridge . . . 129
12.6.6. Injecting beans into the binder . . . 130
12.6.7. Programmatic mapping . . . 130
12.6.8. Incubating features . . . 130

12.7. Declaring dependencies to bridged elements . . . 130
12.7.1. Basics . . . 130
12.7.2. Traversing non-default containers (map keys, …) . . . 131
12.7.3. `reuseRootOnly()`: declaring no dependency at all . . . 134
12.7.4. `fromOtherEntity(…)`: declaring dependencies using the inverse path . . . 134
12.8. Declaring and writing to index fields . . . 136
12.8.1. Basics . . . 136
12.8.2. Type objects . . . 137
12.8.3. Multivalued fields . . . 137
12.8.4. Object fields . . . 138
12.8.5. Object structure . . . 139
12.8.6. Dynamic fields with field templates . . . 139
12.9. Defining index field types . . . 142
12.9.1. Basics . . . 142
12.9.2. Available data types . . . 142
12.9.3. Available type options . . . 142
12.9.4. DSL converter . . . 143
12.9.5. Projection converter . . . 143
12.9.6. Backend-specific types . . . 144
12.10. Defining named predicates . . . 144
12.11. Assigning default bridges with the bridge resolver . . . 147
12.11.1. Basics . . . 147
12.11.2. Assigning a single binder to multiple types . . . 148
12.12. Projection binder . . . 148
12.12.1. Basics . . . 148
12.12.2. Multi-valued projections . . . 149
12.12.3. Composing projection constructors . . . 151
12.12.4. Passing parameters . . . 152
Simple, string parameters . . . 152
Parameters with custom annotations . . . 153
12.12.5. Injecting beans into the binder . . . 154
12.12.6. Programmatic mapping . . . 154
12.12.7. Other incubating features . . . 154
13. Managing the index schema . . . 156
13.1. Basics . . . 156
13.2. Automatic schema management on startup/shutdown . . . 156
13.3. Manual schema management . . . 157
13.4. How schema management works . . . 158
13.5. Exporting the schema . . . 159
13.5.1. Exporting the schema to a set of files . . . 159
13.5.2. Exporting to a custom collector . . . 160
13.5.3. Exporting in offline mode . . . 160
14. Indexing entities . . . 161
14.1. Basics . . . 161
14.2. Indexing plans . . . 161
14.2.1. Basics . . . 161
14.2.2. Synchronization with the indexes . . . 162
Basics . . . 162
Per-session override . . . 163
Custom strategy . . . 163
14.2.3. Indexing plan filter . . . 163
14.3. Implicit, listener-triggered indexing . . . 165
14.3.1. Basics . . . 165
14.3.2. Configuration . . . 166
14.3.3. In-session entity change detection and limitations . . . 166
14.3.4. Dirty checking . . . 166
14.4. Indexing a large amount of data with the `MassIndexer` . . . 166
14.4.1. Basics . . . 166
14.4.2. Selecting types to be indexed . . . 168
14.4.3. Mass indexing multiple tenants . . . 168
14.4.4. Running the mass indexer asynchronously . . . 168

14.4.5. Conditional reindexing . . . 169
14.4.6. `MassIndexer` parameters . . . 169
14.4.7. Tuning the `MassIndexer` for best performance . . . 173
Basics . . . 173
Threads and connections . . . 173
14.5. Indexing a large amount of data with the Jakarta Batch integration . . . 174
14.5.1. Basics . . . 174
14.5.2. Job Parameters . . . 174
14.5.3. Conditional indexing . . . 176
14.5.4. Parallel indexing . . . 176
Threads . . . 176
Rows per partition . . . 177
14.5.5. Chunking and session clearing . . . 177
14.5.6. Selecting the persistence unit (EntityManagerFactory) . . . 178
JBeret . . . 178
Other DI-enabled Jakarta Batch implementations . . . 178
Plain Java environment (no dependency injection at all) . . . 179
14.6. Explicit indexing . . . 179
14.6.1. Basics . . . 179
14.6.2. Configuration . . . 179
14.6.3. Using a `SearchIndexingPlan` manually . . . 179
14.6.4. Hibernate ORM and the periodic "flush-clear" pattern with `SearchIndexingPlan` . . . 181
15. Searching . . . 184
15.1. Query DSL . . . 184
15.1.1. Basics . . . 184
15.1.2. Advanced entity types targeting . . . 184
Targeting multiple entity types . . . 185
Targeting entity types by name . . . 185
15.1.3. Fetching results . . . 185
Basics . . . 185
Fetching all hits . . . 186
Fetching the total (hit count, …) . . . 187
`totalHitCountThreshold(…)`: optimizing total hit count computation . . . 187
Pagination . . . 188
Scrolling . . . 188
15.1.4. Routing . . . 189
15.1.5. Entity loading options for Hibernate ORM . . . 190
Cache lookup strategy . . . 190
Fetch size . . . 190
Entity graph . . . 191
15.1.6. Timeout . . . 192
`failAfter()`: Aborting the query after a given amount of time . . . 192
`truncateAfter()`: Truncating the results after a given amount of time . . . 192
15.1.7. Setting query parameters . . . 193
15.1.8. Obtaining a query object . . . 193
15.1.9. `explain(…)`: Explaining scores . . . 194
15.1.10. `took` and `timedOut`: finding out how long the query took . . . 195
15.1.11. Elasticsearch: leveraging advanced features with JSON manipulation . . . 196
15.1.12. Lucene: retrieving low-level components . . . 197
15.2. Predicate DSL . . . 198
15.2.1. Basics . . . 198
15.2.2. `matchAll`: match all documents . . . 198
`except(…)`: exclude documents matching a given predicate . . . 198
Other options . . . 199
15.2.3. `matchNone`: match no documents . . . 199
15.2.4. `id`: match a document identifier . . . 199
Expected type of arguments . . . 199
Other options . . . 199
15.2.5. `match`: match a value . . . 199
Expected type of arguments . . . 200

Targeting multiple fields . . . 200
Analysis . . . 200
fuzzy: match a text value approximately . . . 201
minimumShouldMatch: fine-tuning how many terms are required to match . . . 201
Other options . . . 202
15.2.6. range: match a range of values . . . 202
Expected type of arguments . . . 203
Targeting multiple fields . . . 203
Other options . . . 204
15.2.7. phrase: match a sequence of words . . . 204
slop: match a sequence of words approximately . . . 204
Targeting multiple fields . . . 204
Other options . . . 204
15.2.8. exists: match fields with content . . . 204
Object fields . . . 205
Other options . . . 205
15.2.9. wildcard: match a simple pattern . . . 205
Targeting multiple fields . . . 206
Other options . . . 206
15.2.10. regexp: match a regular expression pattern . . . 206
Regexp predicates and analysis . . . 206
flags: enabling only specific syntax constructs . . . 207
Targeting multiple fields . . . 207
Other options . . . 208
15.2.11. terms: match a set of terms . . . 208
terms predicates and analysis . . . 208
Expected type of arguments . . . 209
Targeting multiple fields . . . 209
Other options . . . 209
15.2.12. and: match all clauses . . . 209
Adding clauses dynamically with the lambda syntax . . . 209
Options . . . 210
15.2.13. or: match any clause . . . 210
Adding clauses dynamically with the lambda syntax . . . 211
Options . . . 211
15.2.14. not: negating another predicate . . . 211
Other options . . . 212
15.2.15. bool: advanced combinations of predicates (or/and/...) . . . 212
Emulating an OR operator . . . 212
Emulating an AND operator . . . 212
mustNot: excluding documents that match a given predicate . . . 212
filter: matching documents that match a given predicate without affecting the score . . . 213
should as a way to tune scoring . . . 213
minimumShouldMatch: fine-tuning how many should clauses are required to match . . . 214
Adding clauses dynamically with the lambda syntax . . . 214
Deprecated variants . . . 215
Other options . . . 215
15.2.16. simpleQueryString: match a user-provided query string . . . 215
Boolean operators . . . 216
Default boolean operator . . . 216
Prefix . . . 216
Fuzzy . . . 216
Phrase . . . 217
flags: enabling only specific syntax constructs . . . 217
minimumShouldMatch: fine-tuning how many should clauses are required to match . . . 217
Targeting multiple fields . . . 218
Field types and expected format of field values . . . 218
Other options . . . 218
15.2.17. nested: match nested documents . . . 218
Implicit nesting . . . 218

Deprecated variants . . . 219
15.2.18. Within: match points within a circle, box, polygon . . . 219
Matching points within a circle (within a distance to a point) . . . 219
Matching points within a bounding box . . . 220
Matching points within a polygon . . . 220
Targeting multiple fields . . . 221
Other options . . . 221
15.2.19. knn: K-Nearest Neighbors a.k.a. vector search . . . 221
Expected type of arguments . . . 221
Filtering the neighbors . . . 221
Combining knn with other predicates . . . 221
Filtering out irrelevant results with knn similarity . . . 222
Backend specifics and limitations . . . 222
Other options . . . 223
15.2.20. queryString: match a user-provided query string . . . 223
Default boolean operator . . . 223
Phrase slop . . . 223
Allowing leading wildcards . . . 224
Enabling position increments . . . 224
Rewrite method . . . 225
minimumShouldMatch: fine-tuning how many should clauses are required to match . . . 225
Targeting multiple fields . . . 225
Field types and expected format of field values . . . 225
Other options . . . 225
15.2.21. prefix: match documents based on what a field starts with . . . 226
Targeting multiple fields . . . 226
15.2.22. Named: call a predicate defined in the mapping . . . 226
15.2.23. withParameters: create predicates accessing query parameters . . . 226
15.2.24. Backend-specific extensions . . . 227
Lucene: fromLuceneQuery . . . 227
Elasticsearch: fromJson . . . 227
15.2.25. Options common to multiple predicate types . . . 228
Targeting multiple fields in one predicate . . . 228
Tuning the score . . . 229
Overriding analysis . . . 230
15.3. Sort DSL . . . 230
15.3.1. Basics . . . 230
15.3.2. score: sort by matching score (relevance) . . . 231
Options . . . 232
15.3.3. indexOrder: sort according to the order of documents on storage . . . 232
15.3.4. field: sort by field values . . . 232
Syntax . . . 232
Options . . . 232
15.3.5. distance: sort by distance to a point . . . 233
Prerequisites . . . 233
Syntax . . . 233
Options . . . 233
15.3.6. withParameters: create sorts using query parameters . . . 233
15.3.7. composite: combine sorts . . . 233
Adding sorts dynamically with the lambda syntax . . . 234
Stabilizing a sort . . . 235
15.3.8. Backend-specific extensions . . . 235
Lucene: fromLuceneSort . . . 235
Lucene: fromLuceneSortField . . . 235
Elasticsearch: fromJson . . . 236
15.3.9. Options common to multiple sort types . . . 236
Sort order . . . 236
Missing values . . . 237
Sort mode for multivalued fields . . . 238
Filter for fields in nested objects . . . 239

15.4. Projection DSL . . . 239
15.4.1. Basics . . . 239
15.4.2. Projecting to a custom (annotated) type . . . 240
15.4.3. `documentReference`: return references to matched documents . . . 241
Syntax . . . 241
`@DocumentReferenceProjection` in projections to custom types . . . 241
15.4.4. `entityReference`: return references to matched entities . . . 242
Syntax . . . 242
`@EntityReferenceProjection` in projections to custom types . . . 242
15.4.5. `id`: return identifiers of matched entities . . . 243
Syntax . . . 243
`@IdProjection` in projections to custom types . . . 243
15.4.6. `entity`: return matched entities . . . 244
Syntax . . . 244
Requesting a specific entity type . . . 244
`@EntityProjection` in projections to custom types . . . 244
15.4.7. `field`: return field values from matched documents . . . 245
Syntax . . . 245
Multivalued fields . . . 245
Skipping conversion . . . 246
`@FieldProjection` in projections to custom types . . . 246
15.4.8. `score`: return the score of matched documents . . . 247
Syntax . . . 247
`@ScoreProjection` in projections to custom types . . . 247
15.4.9. `distance`: return the distance to a point . . . 248
Syntax . . . 248
Multivalued fields . . . 249
`@DistanceProjection` in projections to custom types . . . 249
15.4.10. `composite`: combine projections . . . 250
Basics . . . 250
Composing more than 3 inner projections . . . 250
Projecting to a `List<?>` or `Object[]` . . . 251
Projecting to a custom (annotated) type . . . 252
`@CompositeProjection` in projections to custom types . . . 253
Deprecated variants . . . 254
15.4.11. `object`: return one value per object in an object field . . . 254
Syntax . . . 254
Composing more than 3 inner projections . . . 255
Projecting to a `List<?>` or `Object[]` . . . 256
Projecting to a custom (annotated) type . . . 257
`@ObjectProjection` in projections to custom types . . . 258
`@ObjectProjection` filters to exclude nested projections and break `@ObjectProjection` cycles . . . 259
15.4.12. `constant`: return a provided constant . . . 262
Syntax . . . 262
In projections to custom types . . . 262
15.4.13. `highlight`: return highlighted field values from matched documents . . . 262
Syntax . . . 262
Multivalued fields . . . 263
Highlighting options . . . 263
`@HighlightProjection` in projections to custom types . . . 264
Highlight limitations . . . 265
15.4.14. `withParameters`: create projections using query parameters . . . 266
Syntax . . . 266
15.4.15. Backend-specific extensions . . . 266
Lucene: `document` . . . 266
Lucene: `documentTree` . . . 267
Lucene: `explanation` . . . 268
Elasticsearch: `source` . . . 268
Elasticsearch: `explanation` . . . 269

Elasticsearch: `jsonHit` ... 269
15.5. Highlight DSL ... 270
15.5.1. Basics ... 270
15.5.2. Highlighter type ... 272
15.5.3. Named highlighters ... 273
15.5.4. Tags ... 274
15.5.5. Encoder ... 275
15.5.6. No match size ... 275
15.5.7. Fragment size and number of fragments ... 276
15.5.8. Order ... 276
15.5.9. Fragmenter ... 277
15.5.10. Boundary scanner ... 277
15.5.11. Phrase limit ... 278
15.6. Aggregation DSL ... 279
15.6.1. Basics ... 279
15.6.2. `terms`: group by the value of a field ... 280
Skipping conversion ... 280
`maxTermCount`: limiting the number of returned entries ... 281
`minDocumentCount`: requiring at least N matching documents per term ... 281
Order of entries ... 282
Aggregated value ... 282
Other options ... 283
15.6.3. `range`: grouped by ranges of values for a field ... 283
Passing `Range` arguments ... 283
Skipping conversion ... 284
Aggregated value ... 285
Other options ... 285
15.6.4. Metric aggregations ... 285
Sum metric aggregation ... 285
Min metric aggregation ... 286
Max metric aggregation ... 286
Count metric aggregation ... 286
Avg metric aggregation ... 287
15.6.5. `composite`: combine aggregations ... 287
Basics ... 287
Composing more than 3 inner aggregations ... 288
Aggregating to a `List<?>` or `Object[]` ... 288
15.6.6. `withParameters`: create aggregations using query parameters ... 290
15.6.7. Backend-specific extensions ... 290
Elasticsearch: `fromJson` ... 290
15.6.8. Options common to multiple aggregation types ... 291
Filter for fields in nested objects ... 291
15.7. Field types and compatibility ... 291
15.7.1. Type of arguments passed to the DSL ... 292
15.7.2. Type of projected values ... 293
15.7.3. Targeting multiple fields ... 294
Incompatible codec ... 294
Incompatible DSL converters ... 295
Incompatible projection converters ... 295
Incompatible analyzer ... 295
15.8. Field paths ... 295
15.8.1. Absolute field paths ... 295
15.8.2. Relative field paths ... 296
16. Explicit backend/index operations ... 298
16.1. Applying configured analyzers/normalizers to a string ... 298
16.2. Explicitly altering a whole index ... 299
16.3. Lucene-specific explicit backend/index operations ... 300
16.3.1. Retrieving analyzers and normalizers through the Lucene-specific `Backend` ... 300
16.3.2. Retrieving the Lucene's index size ... 301
16.3.3. Retrieving a Lucene `IndexReader` ... 301

16.4. Elasticsearch-specific explicit backend/index operations . . . 302
16.4.1. Retrieving the REST client . . . 302
17. Lucene backend . . . 303
17.1. Basic configuration . . . 303
17.2. Index storage (`Directory`) . . . 303
17.2.1. Local filesystem storage . . . 303
Index location . . . 303
Filesystem access strategy . . . 303
Other configuration options . . . 304
17.2.2. Local heap storage . . . 304
17.2.3. Locking strategy . . . 304
17.3. Sharding . . . 305
17.3.1. Basics . . . 305
17.3.2. Per-shard configuration . . . 305
17.4. Index format compatibility . . . 306
17.5. Schema . . . 306
17.5.1. Field types . . . 306
Available field types . . . 306
Index field type DSL extensions . . . 308
17.5.2. Multi-tenancy . . . 309
`none`: single-tenancy . . . 309
`discriminator`: type name mapping using the index name . . . 309
17.6. Analysis . . . 309
17.6.1. Basics . . . 309
17.6.2. Built-in analyzers . . . 310
17.6.3. Built-in normalizers . . . 310
17.6.4. Custom analyzers and normalizers . . . 310
Referencing components by name . . . 310
Referencing components by factory class . . . 311
Assigning names to analyzer instances . . . 312
17.6.5. Overriding the default analyzer . . . 312
17.6.6. Similarity . . . 313
17.7. Threads . . . 313
17.8. Indexing queues . . . 313
17.9. Writing and reading . . . 314
17.9.1. Commit . . . 314
17.9.2. Refresh . . . 315
17.9.3. `IndexWriter` settings . . . 315
17.9.4. Merge settings . . . 316
17.10. Searching . . . 317
17.10.1. Low-level hit caching . . . 317
18. Elasticsearch backend . . . 319
18.1. Compatibility . . . 319
18.1.1. Overview . . . 319
18.1.2. Elasticsearch . . . 319
18.1.3. OpenSearch . . . 319
18.1.4. Amazon OpenSearch Service . . . 319
18.1.5. Amazon OpenSearch Serverless (incubating) . . . 319
18.1.6. Upgrading Elasticsearch . . . 320
18.2. Basic configuration . . . 320
18.3. Configuration of the Elasticsearch cluster . . . 320
18.4. Client configuration . . . 320
18.4.1. Available clients . . . 320
Elasticsearch low level REST client (Apache HTTP Client 4) . . . 320
Elasticsearch low level REST client (Apache HTTP Client 5) . . . 321
OpenSearch low level REST client (Apache HTTP Client 5) . . . 321
Simple REST client based on the JDKs `HttpClient` . . . 321
18.4.2. Target hosts . . . 322
18.4.3. Path prefix . . . 323
18.4.4. Node discovery . . . 323
18.4.5. HTTP authentication . . . 323

18.4.6. Authentication on Amazon Web Services . . . 324
18.4.7. Connection tuning . . . 325
18.4.8. Custom HTTP client configurations . . . 326
18.4.9. Elasticsearch low level client . . . 326
18.4.10. Elasticsearch java client . . . 327
18.4.11. OpenSearch low level client . . . 328
18.4.12. JDK `HttpClient` based REST client . . . 329
18.5. Version compatibility . . . 329
18.5.1. Version assumed by Hibernate Search . . . 329
18.5.2. Disabling the version check on startup . . . 330
18.6. Request logging . . . 330
18.7. Sharding . . . 330
18.8. Schema management . . . 330
18.9. Index layout . . . 330
18.9.1. `simple`: the default, future-proof strategy . . . 331
18.9.2. `no-alias`: a strategy without index aliases . . . 331
18.9.3. Custom strategy . . . 331
18.9.4. Retrieving index or alias names . . . 332
18.10. Schema ("mapping") . . . 333
18.10.1. Field types . . . 333
Available field types . . . 333
Index field type DSL extension . . . 334
18.10.2. Entity type name mapping . . . 335
`discriminator`: type name mapping using a discriminator field . . . 335
`index-name`: type name mapping using the index name . . . 336
18.10.3. Dynamic mapping . . . 336
18.10.4. Multi-tenancy . . . 336
`none`: single-tenancy . . . 336
`discriminator`: type name mapping using the index name . . . 337
18.10.5. Custom index mapping . . . 337
Basics . . . 337
Disabling `_source` . . . 338
18.11. Analysis . . . 338
18.11.1. Basics . . . 338
18.11.2. Built-in analyzers . . . 339
18.11.3. Built-in normalizers . . . 340
18.11.4. Custom analyzers and normalizers . . . 340
18.11.5. Overriding the default analyzer . . . 341
18.12. Custom index settings . . . 341
18.12.1. Max result window size . . . 342
18.13. Threads . . . 342
18.14. Indexing queues . . . 342
18.15. Writing and reading . . . 343
18.15.1. Commit . . . 343
18.15.2. Refresh . . . 344
18.16. Searching . . . 344
18.16.1. Scroll timeout . . . 344
18.16.2. Partial shard failure . . . 344
19. Coordination . . . 345
19.1. Basics . . . 345
19.2. No coordination . . . 345
19.2.1. Basics . . . 345
19.2.2. How indexing works without coordination . . . 345
19.3. `outbox-polling`: additional event tables and polling in background processors . . . 346
19.3.1. Basics . . . 346
19.3.2. How indexing works with `outbox-polling` coordination . . . 347
19.3.3. Impact on the database schema . . . 348
Basics . . . 348
Custom schema/table name/etc . . . 348
19.3.4. Sharding and pulse . . . 349

19.3.5. Event processor . . . 350
Basics . . . 350
Sharding . . . 351
Processing order . . . 352
19.3.6. Mass indexer . . . 352
Basics . . . 352
19.3.7. Multi-tenancy . . . 353
19.3.8. Aborted events . . . 354
20. Static metamodel . . . 355
20.1. Overview of the static metamodel . . . 355
20.1.1. Field reference types . . . 355
20.2. Annotation processor . . . 356
20.2.1. Enabling the annotation processor . . . 356
20.2.2. Configuration . . . 357
20.2.3. Current annotation processor limitations . . . 358
21. Standards and integrations . . . 359
21.1. Jakarta EE . . . 359
21.2. Java EE . . . 359
21.3. Hibernate ORM 6/5 . . . 359
21.4. Lucene 10 . . . 359
22. Known issues and limitations . . . 360
22.1. Without coordination, in rare cases, indexing involving `@IndexedEmbedded` may lead to out-of sync indexes . . . 360
22.1.1. Description . . . 360
22.1.2. Solutions and workarounds . . . 360
22.1.3. Roadmap . . . 360
22.2. Without coordination, backend errors during indexing may lead to out-of sync indexes . . . 360
22.2.1. Description . . . 360
22.2.2. Solutions and workarounds . . . 360
22.2.3. Roadmap . . . 361
22.3. Listener-triggered indexing only considers changes applied directly to entity instances in Hibernate ORM sessions . . . 361
22.3.1. Description . . . 361
22.3.2. Solutions and workarounds . . . 361
22.3.3. Roadmap . . . 361
22.4. Listener-triggered indexing ignores asymmetric association updates . . . 361
22.4.1. Description . . . 361
22.4.2. Solutions and workarounds . . . 361
22.4.3. Roadmap . . . 362
22.5. Listener-triggered indexing is not compatible with `Session` serialization . . . 362
22.5.1. Description . . . 362
22.5.2. Solutions and workarounds . . . 362
22.5.3. Roadmap . . . 362
23. Troubleshooting . . . 363
23.1. Finding out what is executed under the hood . . . 363
23.2. Loggers . . . 363
23.3. Frequently asked questions . . . 363
23.3.1. Unexpected or missing documents in search hits . . . 363
23.3.2. Unsatisfying order of search hits when sorting by score . . . 363
23.3.3. Search query execution takes too long . . . 364
24. Further reading . . . 365
24.1. Hibernate Search . . . 365
24.2. Elasticsearch . . . 365
24.3. Lucene . . . 365
24.4. Hibernate ORM . . . 365
24.5. Other . . . 365
25. Credits . . . 366
Appendix A: List of all available configuration properties . . . 367
A.1. Hibernate Search Engine . . . 367
A.2. Hibernate Search Backend - Lucene . . . 367
A.3. Hibernate Search Backend - Elasticsearch . . . 373
A.4. Hibernate Search Backend - Elasticsearch - AWS integration . . . 380
A.5. Hibernate Search Backend - Elasticsearch client based on the low-level Elasticsearch REST client . . . 381

A.6. Hibernate Search Backend - Elasticsearch client based on the low-level Elasticsearch REST client . . . 383
A.7. Hibernate Search Backend - Elasticsearch client based on the low-level OpenSearch client . . . 385
A.8. Hibernate Search ORM Integration . . . 386
A.9. Hibernate Search ORM Integration - Coordination - Outbox Polling . . . 389
A.10. Hibernate Search Mapper - POJO Standalone . . . 394
Appendix B: List of all available logging categories . . . 396

Preface

Full text search engines like Apache Lucene are very powerful technologies to add efficient free text search capabilities to applications. However, Lucene suffers several mismatches when dealing with object domain models. Amongst other things indexes have to be kept up to date and mismatches between index structure and domain model as well as query mismatches have to be avoided.

Hibernate Search addresses these shortcomings: it indexes your domain model with the help of a few annotations, takes care of database/index synchronization and brings back regular managed objects from free text queries.

To achieve this, Hibernate Search combines the power of Hibernate ORM and Apache Lucene/Elasticsearch/OpenSearch.

Chapter 1. Compatibility

1.1. Dependencies

Table 1.1: Compatible versions of dependencies

	Version	Note
Java Runtime	17, 21 or 25	
Hibernate ORM (for the Hibernate ORM mapper)	7.4.0.CR1	
Jakarta Persistence (for the Hibernate ORM mapper)	3.2	
Apache Lucene (for the Lucene backend)	9.12.3	Or Lucene 10.4.0 when using JDK 21+ and different Maven artifacts.
Elasticsearch server (for the Elasticsearch backend)	7.10+, 8.x or 9.x	Most of older minor versions (e.g. 7.11 or 8.0) are not given priority for bugfixes and new features.
OpenSearch server (for the Elasticsearch backend)	1.3, 2.x or 3.x	Other minor versions may work but are not given priority for bugfixes and new features.

Find more information for all versions of Hibernate Search on our compatibility matrix.

The compatibility policy may also be of interest.

Elasticsearch licensing

While Elasticsearch up to 7.10 was distributed under the Apache License 2.0, be aware that Elasticsearch 7.11-8.15 versions are distributed under the Elastic License and the SSPL, which are not considered open-source by the Open Source Initiative. Starting with Elasticsearch 8.16 the AGPL v3 license was added. Please refer to https://www.elastic.co/ to learn more on the licensing of Elasticsearch.

Only the low-level Java REST client, which Hibernate Search depends on, remains open-source.

OpenSearch

While it historically targeted Elastic's Elasticsearch distribution, Hibernate Search is also compatible with OpenSearch and regularly tested against it; see Compatibility for more information.

Every section of this documentation referring to Elasticsearch is also relevant for the OpenSearch distribution.

1.1.1. Adding Hibernate Search dependencies to your project

Hibernate Search BOM

If you get Hibernate Search from Maven, it is recommended to import Hibernate Search BOM as part of your dependency management to keep all its artifact versions aligned:

```
<dependencyManagement>
    <dependencies>
        <!--
            Import Hibernate Search BOM
            to get all of its artifact versions aligned:
        -->
        <dependency>
            <groupId>org.hibernate.search</groupId>
            <artifactId>hibernate-search-bom</artifactId>
            <version>8.4.0.CR1</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
```

```
        <!-- Any other dependency management entries -->
    </dependencies>
</dependencyManagement>
```

Hibernate Search Platform

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

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

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

Besides the lean BOM file, Hibernate Search also provides several platform POM files that manage the versions of Hibernate Search artifacts, *and* their transitive dependencies, *and* related artifacts that must be aligned. For example, it brings the management of all other `org.hibernate.orm` artifacts, beyond the ones required by the ORM mapper, or extra Lucene artifacts like `lucene-suggest` or `lucene-analysis-icu` and others, that can be helpful for more advanced applications. These platform files will help keep the versions of extra Hibernate ORM/Lucene/Elasticsearch client dependencies aligned with the versions of artifacts from the same groups that are used by Hibernate Search itself.

Currently, there are two platform POM files for Hibernate Search:

- `hibernate-search-platform-bom`: use this when in doubt.
- `hibernate-search-platform-next-bom`: use this if you want to use the Lucene-next backend.

To leverage the dependency management provided by these platform files, use the same approach of importing as for the regular BOM file:

Importing the Hibernate Search platform BOM

```
<dependencyManagement>
    <dependencies>
        <!--
            Import Hibernate Search platform
            to get all of its artifact versions aligned:
        -->
        <dependency>
            <groupId>org.hibernate.search</groupId>
            <artifactId>hibernate-search-platform-bom</artifactId>
            <version>8.4.0.CR1</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <!-- Any other dependency management entries -->
    </dependencies>
</dependencyManagement>

<dependencies>
    <!-- Any other dependency management entries -->
    <!--
         For example, add an extra Lucene dependency without specifying the version
         as it is managed by the platform POM:
    -->
    <dependency>
        <groupId>org.apache.lucene</groupId>
        <artifactId>lucene-suggest</artifactId>
    </dependency>
    <!-- Any other dependency management entries -->
</dependencies>
```

1.2. Framework support

1.2.1. Quarkus

Quarkus has an official extension for Hibernate Search with Hibernate ORM using the Elasticsearch backend, which is a tight integration with additional features, different dependencies, and different configuration properties.

As your first step to using Hibernate Search within Quarkus, we recommend you follow Quarkus's Hibernate Search Guide: it is a great

hands-on introduction to Hibernate Search, *and* it covers the specifics of Quarkus.

1.2.2. WildFly

WildFly includes modules for Hibernate Search with Hibernate ORM using either the Lucene backend or the Elasticsearch backend.

To start using Hibernate Search within WildFly, see the Hibernate Search section in the WildFly Developer Guide: it covers all the specifics of WildFly.

1.2.3. Spring Boot

Hibernate Search can easily be integrated into a Spring Boot application. Just read about Spring Boot's specifics below, then follow the getting started guide.

Configuration properties

`application.properties`/`application.yaml` are Spring Boot configuration files, not JPA or Hibernate Search configuration files. Adding Hibernate Search properties starting with `hibernate.search.` directly in that file will not work.

When integrating Hibernate Search with Hibernate ORM

Prefix your Hibernate Search properties with `spring.jpa.properties.`, so that Spring Boot passes along the properties to Hibernate ORM, which will pass them along to Hibernate Search.

For example:

```
spring.jpa.properties.hibernate.search.backend.hosts = elasticsearch.mycompany.com
```

When using the Standalone POJO mapper

You can pass properties programmatically to `SearchMappingBuilder#property`.

Dependency versions

Spring Boot automatically sets the version of dependencies without your knowledge. While this is ordinarily a good thing, from time to time Spring Boot dependencies will be a little out of date. Thus, it is recommended to override Spring Boot's defaults at least for some key dependencies.

With Maven, there are a few ways to override these versions depending on how Spring is added to the application. If your application's POM file is using `spring-boot-starter-parent` as its parent POM then simply adding version properties to your POM's `<properties>` should help:

```
<properties>
    <hibernate.version>7.4.0.CR1</hibernate.version>
    <elasticsearch-client.version>9.4.0</elasticsearch-client.version>
    <!-- ... plus any other properties of yours ... -->
</properties>
```

If, after setting the properties above, you still are getting the same version of the libraries, check if property names in the Spring Boot's BOM have changed, and if so use the new property name.

Alternatively, if either the `spring-boot-dependencies` or the `spring-boot-starter-parent` is imported into the dependency management (`<dependencyManagement>`) then overriding the versions can be done either by importing a BOM listing the dependencies we want to override, or by explicitly listing a dependency with its version that we want to be used:

Override dependencies either with another BOM or explicitly

```
<dependencyManagement>
    <dependencies>
        <!--
            Overriding Hibernate ORM version by importing the BOM.
            Alternatively, can be done by adding specific dependencies
            as shown below for Elasticsearch dependencies.
        -->
        <dependency>
            <groupId>org.hibernate.orm</groupId>
            <artifactId>hibernate-platform</artifactId>
            <version>7.4.0.CR1</version>
```

```
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-dependencies</artifactId>
            <version>4.0.1</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <!--
            Since there is no BOM for the Elasticsearch REST client,
            these dependencies have to be listed explicitly:
        -->
        <dependency>
            <groupId>org.elasticsearch.client</groupId>
            <artifactId>elasticsearch-rest-client</artifactId>
            <version>9.4.0</version>
        </dependency>
        <dependency>
            <groupId>org.elasticsearch.client</groupId>
            <artifactId>elasticsearch-rest-client-sniffer</artifactId>
            <version>9.4.0</version>
        </dependency>
        <!-- Other dependency management entries -->
    </dependencies>
</dependencyManagement>
```

For other build tools refer to their documentation for details.

Maven's `dependency` plugin (or your build tool corresponding alternative) can be used to verify that the version override was correctly applied, e.g.:

```
# Show the dependency tree filtering for Hibernate and Elasticsearch dependencies to reduce the
output:
mvn dependency:tree "-Dincludes=org.hibernate.*,org.elasticsearch.*"
```

If, after setting the properties above, you still have problems (e.g. `NoClassDefFoundError`) with some of Hibernate Search's dependencies, look for the version of that dependency in Spring Boot's POM and Hibernate Search's POM: there will probably be a mismatch, and generally overriding Spring Boot's version to match Hibernate Search's version will work fine.

Application hanging on startup

Spring Boot 2.3.x and above is affected by a bug that causes the application to hang on startup when using Hibernate Search, particularly when using custom components (custom bridges, analysis configurers, …).

The problem, which is not limited to just Hibernate Search, has been reported, but hasn't been fixed yet in Spring Boot 2.5.1.

As a workaround, you can set the property `spring.data.jpa.repositories.bootstrap-mode` to `deferred` or, if that doesn't work, `default`. Interestingly, using `@EnableJpaRepositories(bootstrapMode = BootstrapMode.DEFERRED)` has been reported to work even in situations where setting `spring.data.jpa.repositories.bootstrap-mode` to `deferred` didn't work.

Alternatively, if you do not need dependency injection in your custom components, you can refer to those components with the prefix `constructor:` so that Hibernate Search doesn't even try to use Spring to retrieve the components, and thus avoids the deadlock in Spring. See this section for more information.

Spring Boot's Elasticsearch client and auto-configuration

As you may know, Spring Boot includes "auto-configuration" that triggers as soon as a dependency is detected in the classpath.

This may lead to problems in some cases when dependencies are used by the application, but not through Spring Boot.

In particular, Hibernate Search transitively brings in a dependency to Elasticsearch's low-level REST Client. Spring Boot, through `ElasticsearchRestClientAutoConfiguration`, will automatically set up an Elasticsearch REST client targeting (by default) `http://localhost:9200` as soon as it detects that dependency to the Elasticsearch REST Client JAR.

If your Elasticsearch cluster is not reachable at `http://localhost:9200`, this might lead to errors on startup.

To get rid of these errors, either configure Spring's Elasticsearch client manually, or disable this specific auto-configuration.

Spring Boot's Elasticsearch client is completely separate from Hibernate Search: the configuration of one won't affect the other.

1.2.4. Other

If your framework of choice is not mentioned in the previous sections, don't worry: Hibernate Search works just fine with plenty of other frameworks.

Just follow the getting started guide to try it out.

Chapter 2. Getting started with Hibernate Search

To get started with Hibernate Search, check out the following guides:

- If your entities **are** defined in Hibernate ORM, see Getting started with Hibernate Search in Hibernate ORM.
- If your entities are **not** defined in Hibernate ORM, see Getting started with Hibernate Search's Standalone POJO Mapper instead.

Chapter 3. Migrating

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

To Hibernate Search 5 users

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

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

Additionally, be aware that a lot of APIs have changed, some only because of a package change, others because of more fundamental changes (like moving away from using Lucene types in Hibernate Search APIs). For that reason, you are encouraged to migrate first to Hibernate Search 6.0 using the 6.0 migration guide, and only then to later versions (which will be significantly easier).

Chapter 4. Concepts

4.1. Full-text search

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

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

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

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

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

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

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

4.2. Entity types

When it comes to the domain model of applications, Hibernate Search distinguishes between types (Java classes) that are considered entities, and those that are not.

The defining characteristic of entity types in Hibernate Search is that their instances have a distinct lifecycle: an entity instance may be saved into a datastore, or retrieved from it, without requiring the saving or retrieval of an instance of another type. For that purpose, each entity instance is assumed to carry an immutable, unique identifier.

These characteristics allow Hibernate Search to map entity types to indexes, but only entity types. "Embeddable" types that are referenced from or contained within an entity, but whose lifecycle is completely tied to that entity, cannot be mapped to an index.

Multiple aspects of Hibernate Search involve the concept of entity types:

1. Each entity type has an entity name, distinct from the type name. E.g. for a class named `com.acme.Book`, the entity name could be `Book` (the default), or any arbitrarily chosen string.
2. Properties pointing to an entity type (called *associations*) have specific mechanics; in particular, in order to handle reindexing, Hibernate Search needs to know about the inverse side of associations.
3. For the purposes of change tracking when reindexing, (e.g. in indexing plans), entity types represent the smallest scope Hibernate Search considers.

   This means the paths representing "changed properties" in Hibernate Search always have an entity as their starting point, and the components within these paths never reach into another entity (but may point to one, when an *association* changes.

4. Hibernate Search may need additional configuration to enable loading of entity types from an external datastore, be it to load entities matching a query from an external source or to load all entity instances from an external source for full reindexing.

4.3. Mapping

Applications targeted by Hibernate search use an entity-based model to represent data. In this model, each entity is a single object with a few properties of atomic type (`String`, `Integer`, `LocalDate`, …). Each entity can contain non-root aggregates ("embeddable" types), and each can also have multiple associations to one or even many other entities.

By contrast, Lucene and Elasticsearch work with documents. Each document is a collection of "fields", each field being assigned a name—a unique string—and a value—which can be text, but also numeric data such as an integer or a date. Fields also have a type, which not only determines the type of values (text/numeric), but more importantly the way this value will be stored: indexed, stored, with doc values, etc. Each document can contain nested aggregates ("objects"/"nested documents"), but there cannot really be associations between top-level documents.

Thus:

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

- Documents are organized, at best, as a collection of trees, where each tree is a document, optionally with nested documents.

There are multiple mismatches between the entity model and the document model: simple property types vs. more complex field types, associations vs. no associations, graph vs. collection of trees.

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

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

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

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

4.4. Binding

While the mapping definition is declarative, these declarations need to be interpreted and actually applied to the domain model.

That's what Hibernate Search calls "binding": during startup, a given mapping instruction (e.g. `@GenericField`) will result in a "binder" being instantiated and called, giving it an opportunity to inspect the part of the domain model it's applied to and to "bind" (assign) a component to that part of the model — for example a "bridge", responsible for extracting data from an entity during indexing.

Hibernate Search comes with binders and bridges for many common use cases, and also provides the ability to plug in custom binders and bridges.

For more information, in particular on how to plug in custom binders and bridges, see Binding and bridges.

4.5. Analysis

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

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

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

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

1. Character filter: transforms the input characters. Replaces, adds or removes characters.
2. Tokenizer: splits the text into several words, called "tokens".
3. Token filter: transforms the tokens. Replaces, add or removes characters in a token, derives new tokens from the existing ones, removes tokens based on some condition, …

The tokenizer usually splits on whitespaces (though there are other options). Token filters are usually where customization takes place. They can remove accented characters, remove meaningless suffixes (`-ing`, `-s`, …) or tokens (`a`, `the`, …), replace tokens with a chosen spelling (`wi-fi` ⇒ `wifi`), etc.

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

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

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

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

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

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

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

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

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

4.6. Commit and refresh

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

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

  The changes are "pushed" to the actual index when the writer is *committed*. Until the commit happens, uncommitted changes are in an "unsafe" state: if the application crashes or if the server suffers from a power loss, uncommitted changes will be lost.

- When reading, e.g. when executing a search query, data is not read *directly* from the index, but from an "index reader" that exposes a view of the index as it was at some point in the past.

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

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

Different factors influence when refreshes and commit happen:

- Listener-triggered indexing and explicit indexing will, by default, require that a commit of the index writer is performed after each set of changes, meaning the changes are safe after the Hibernate ORM transaction commit returns (for the Hibernate ORM integration) or the `SearchSession`'s `close()` method returns (for the Standalone POJO Mapper). However, no refresh is requested by default, meaning the changes may only be visible at a later time, when the backend decides to refresh the index reader. This behavior can be customized by setting a different synchronization strategy.
- The mass indexer will not require any commit or refresh until the very end of mass indexing, to maximize indexing throughput.
- Whenever there are no particular commit or refresh requirements, backend defaults will apply:
  - See here for Elasticsearch.
  - See here for Lucene.
- A commit may be forced explicitly through the `flush()` API.
- A refresh may be forced explicitly though the `refresh()` API.

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

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

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

4.7. Sharding and routing

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

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

When indexing:

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

When searching:

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

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

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

To enable sharding, some configuration is required:

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

Sharding is static by nature: each index is expected to have the same shards, with the same identifiers, from one boot to the other. Changing the number of shards or their identifiers will require full reindexing.

Chapter 5. Architecture

5.1. Components of Hibernate Search

From the user's perspective, Hibernate Search consists of two components:

Mapper

The mapper "maps" the user model to an index model, and provide APIs consistent with the user model to perform indexing and searching.

Most applications rely on the Hibernate ORM mapper, which offers the ability to index properties of Hibernate ORM entities, but there is also a Standalone POJO mapper that can be used without Hibernate ORM.

The mapper is configured partly through annotations on the domain model, and partly through configuration properties.

Backend

The backend is the abstraction over the full-text engines, where "things get done". It implements generic indexing and searching interfaces for use by the mapper through "index managers", each providing access to one index.

For instance the Lucene backend delegates to the Lucene library, and the Elasticsearch backend delegates to a remote Elasticsearch cluster.

The backend is configured partly by the mapper, which tells the backend which indexes must exist and what fields they must have, and partly through configuration properties.

The mapper and the backend work together to provide three main features:

Mass indexing

This is how Hibernate Search rebuilds indexes from zero based on the content of a database.

The mapper queries the database to retrieve the identifier of every entity, then processes these identifiers in batches, loading the entities then processing them to generate documents that are sent to the backend for indexing. The backend puts the document in an internal queue, and will index documents in batches, in background processes, notifying the mapper when it's done.

See Indexing a large amount of data with the `MassIndexer` for details.

Explicit and listener-triggered indexing

Explicit and listener-triggered indexing rely on indexing plans (`SearchIndexingPlan`) to index specific entities as a result of limited changes.

With explicit indexing, the caller explicitly passes information about changes on entities to an indexing plan; with listener-triggered indexing, entity changes are detected transparently by the Hibernate ORM integration (with a few exceptions) and added to the indexing plan automatically.

Listener-triggered indexing only makes sense in the context of the Hibernate ORM integration; there is no such feature available for the Standalone POJO Mapper.

In both cases, the indexing plan will deduce from those changes whether entities need to be reindexed, be it the changed entity itself or other entities that embed the changed entity in their index.

Upon transaction commit, changes in the indexing plan are processed (either in the same thread or in a background process, depending on the coordination strategy), and documents are generated, then sent to the backend for indexing. The backend puts the documents in an internal queue, and will index documents in batches, in background processes, notifying the mapper when it's done.

See Implicit, listener-triggered indexing for details.

Searching

This is how Hibernate Search provides ways to query an index.

The mapper exposes entry points to the search DSL, allowing selection of entity types to query. When one or more entity types are selected, the mapper delegates to the corresponding index managers to provide a Search DSL and ultimately create the search query. Upon query execution, the backend submits a list of entity references to the mapper, which loads the corresponding entities. The entities are then returned by the query.

See Searching for details.

5.2. Examples of architectures

5.2.1. Overview

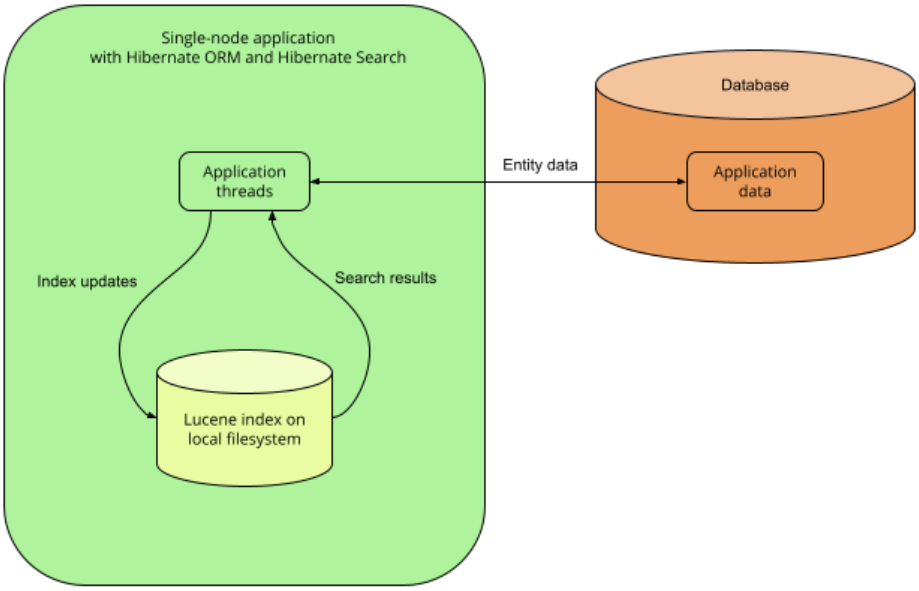
Table 5.1: Comparison of architectures

Architecture	Single-node with Lucene	No coordination with Elasticsearch	Outbox polling with Elasticsearch
Compatible mappers	Both Hibernate ORM integration and Standalone POJO Mapper		Hibernate ORM integration only
Application topology	Single-node	Single-node or multi-node	
Extra bits to maintain	Indexes on filesystem	Elasticsearch cluster	
Guarantee of index updates	Non-transactional, after the database transaction / `SearchSession.close()` returns		Transactional, on database transaction commit
Visibility of index updates	Configurable: immediate or eventual	Configurable: immediate (poor performance) or eventual	Eventual
Native features	Mostly for experts	For anyone	
Overhead for application threads	Low to medium		Very low
Overhead for the database	Low		Low to medium
Impact on database schema	None		Extra tables
Limitations	Listener-triggered indexing ignores: JPQL/SQL queries, asymmetric association updates		
	Out-of-sync indexes in rare situations: concurrent `@IndexedEmbedded`, backend I/O errors		No other known limitation

5.2.2. Single-node application with the Lucene backend

Description

With the Lucene backend, indexes are local to a given application node (JVM). They are accessed through direct calls to the Lucene library, without going through the network.



This mode is only relevant to single-node applications.

Pros and cons

Pros:

- Simplicity: no external services are required, everything lives on the same server.
- Immediate visibility (~milliseconds) of index updates. While other architectures can perform comparably well for most use cases, a single-node, Lucene backend is the best way to implement indexing if you need changes to be visible immediately after the database changes.

Cons:

- Without coordination, backend errors during indexing may lead to out-of sync indexes.
- Without coordination, in rare cases, indexing involving `@IndexedEmbedded` may lead to out-of sync indexes.
- Not so easy to extend: experienced developers can access a lot of Lucene features, even those that are not exposed by Hibernate Search, by providing native Lucene objects; however, Lucene APIs are not very easy to figure out for developers unfamiliar with Lucene. If you're interested, see for example `Query`-based predicates.
- Overhead for application threads: reindexing is done directly in application threads, and it may require additional time to load data that must be indexed from the database. Depending on the amount of data to load, this may increase the application's latency and/or decrease its throughput.
- No horizontal scalability: there can only be one application node, and all indexes need to live on the same server.

Getting started

To implement this architecture, use the following Maven dependencies:

When integrating with Hibernate ORM

```
<dependency>
   <groupId>org.hibernate.search</groupId>
   <artifactId>hibernate-search-mapper-orm</artifactId>
   <version>8.4.0.CR1</version>
</dependency>
<dependency>
   <groupId>org.hibernate.search</groupId>
   <artifactId>hibernate-search-backend-lucene</artifactId>
   <version>8.4.0.CR1</version>
</dependency>
```

With the Standalone POJO Mapper (no Hibernate ORM)

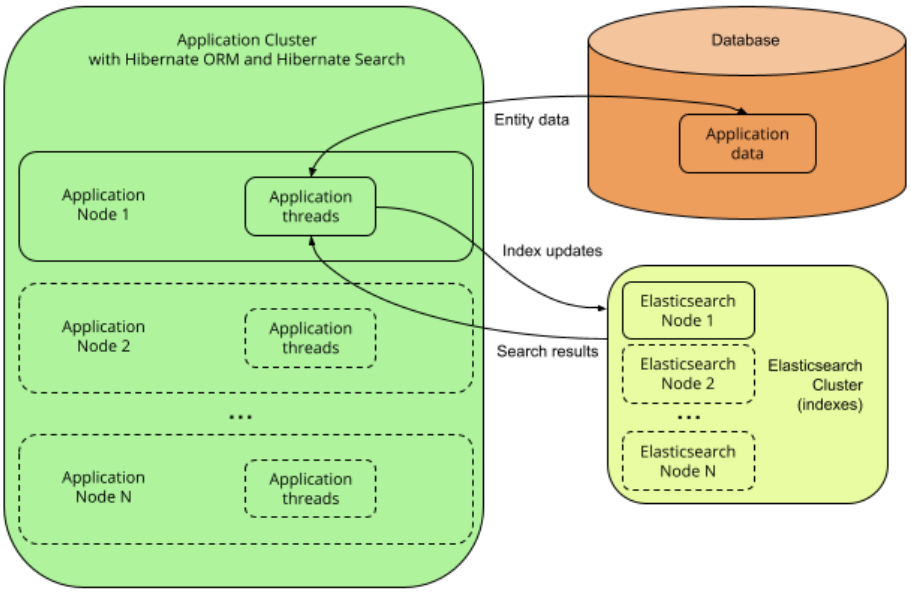
```
<dependency>
   <groupId>org.hibernate.search</groupId>
   <artifactId>hibernate-search-mapper-pojo-standalone</artifactId>
   <version>8.4.0.CR1</version>
</dependency>
<dependency>
   <groupId>org.hibernate.search</groupId>
   <artifactId>hibernate-search-backend-lucene</artifactId>
   <version>8.4.0.CR1</version>
</dependency>
```

5.2.3. Single-node or multi-node application, without coordination and with the Elasticsearch backend

Description

With the Elasticsearch backend, indexes are not tied to the application node. They are managed by a separate cluster of Elasticsearch nodes, and accessed through calls to REST APIs.

Thus, it is possible to set up multiple application nodes in such a way that they all perform index updates and search queries independently, without coordinating with each other.



The Elasticsearch cluster may be a single node living on the same server as the application.

Pros and cons

Pros:

- Easy to extend: you can easily access most Elasticsearch features, even those that are not exposed by Hibernate Search, by providing your own JSON. See for example JSON-defined predicates, or JSON-defined aggregations, or leveraging advanced features with JSON manipulation.
- Horizontal scalability of the indexes: you can size the Elasticsearch cluster according to your needs. See "Scalability and resilience" in the Elasticsearch documentation.
- Horizontal scalability of the application: you can have as many instances of the application as you need (though high concurrency increases the likeliness of some problems with this architecture, see "Cons" below).

Cons:

- Without coordination, backend errors during indexing may lead to out-of sync indexes.
- Without coordination, in rare cases, indexing involving `@IndexedEmbedded` may lead to out-of sync indexes.
- Need to manage an additional service: the Elasticsearch cluster.
- Overhead for application threads: reindexing is done directly in application threads, and it may require additional time to load data that must be indexed from the database. Depending on the amount of data to load, this may increase the application's latency and/or decrease its throughput.
- Delayed visibility (~1 second) of index updates (near-real-time). While changes can be made visible as soon as possible after the database changes, Elasticsearch is near-real-time by nature, and won't perform very well if you need changes to be visible immediately after the database changes.

Getting started

To implement this architecture, use the following Maven dependencies:

When integrating with Hibernate ORM

```
<dependency>
   <groupId>org.hibernate.search</groupId>
   <artifactId>hibernate-search-mapper-orm</artifactId>
   <version>8.4.0.CR1</version>
</dependency>
<dependency>
   <groupId>org.hibernate.search</groupId>
   <artifactId>hibernate-search-backend-elasticsearch</artifactId>
   <version>8.4.0.CR1</version>
</dependency>
```

With the Standalone POJO Mapper (no Hibernate ORM)

```
<dependency>
   <groupId>org.hibernate.search</groupId>
   <artifactId>hibernate-search-mapper-pojo-standalone</artifactId>
   <version>8.4.0.CR1</version>
</dependency>
<dependency>
   <groupId>org.hibernate.search</groupId>
   <artifactId>hibernate-search-backend-elasticsearch</artifactId>
   <version>8.4.0.CR1</version>
</dependency>
```

5.2.4. Multi-node application with outbox polling and Elasticsearch backend

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

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

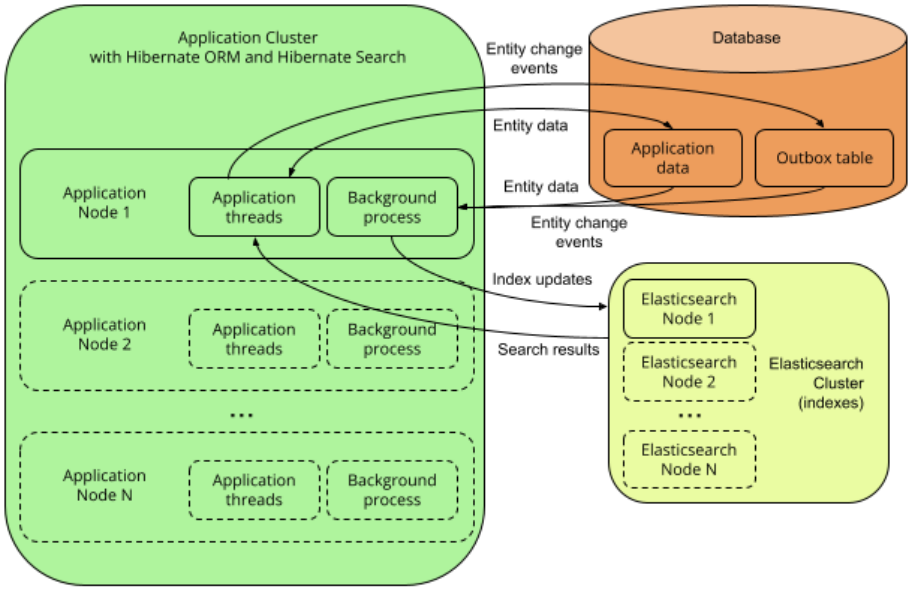
You are encouraged to use incubating features so the development team can get feedback and improve them, but you should be prepared to update code which relies on them as needed.

Description

With Hibernate Search's `outbox-polling` coordination strategy, entity change events are not processed immediately in the ORM session where they arise, but are pushed to an outbox table in the database.

A background process polls that outbox table for new events, and processes them asynchronously, updating the indexes as necessary. Since that queue can be sharded, multiple application nodes can share the workload of indexing.

This requires the Elasticsearch backend so that indexes are not tied to a single application node and can be updated or queried from multiple application nodes.



Pros and cons

Pros:

- Safest:
  - the possibility of out-of-sync indexes caused by indexing errors in the backend that affects other architectures is eliminated here, because entity change events are persisted in the same transaction as the entity changes allowing retries for as long as necessary.
  - the possibility of out-of-sync indexes caused by concurrent updates that affects other architectures is eliminated here, because each entity instance is reloaded from the database within a new transaction before being re-indexed.

- Easy to extend: you can easily access most Elasticsearch features, even those that are not exposed by Hibernate Search, by providing your own JSON. See for example JSON-defined predicates, or JSON-defined aggregations, or leveraging advanced features with JSON manipulation.
- Minimal overhead for application threads: application threads only need to append events to the queue, they don't perform reindexing themselves.
- Horizontal scalability of the indexes: you can size the Elasticsearch cluster according to your needs. See "Scalability and resilience" in the Elasticsearch documentation.
- Horizontal scalability of the application: you can have as many instances of the application as you need.

Cons:

- Need to manage an additional service: the Elasticsearch cluster.
- Delayed visibility (~1 second or more, depending on load and hardware) of index updates. First because Elasticsearch is near-real-time by nature, but also because the event queue introduces additional delays.
- Impact on the database schema: additional tables must be created in the database to hold the data necessary for coordination.
- Overhead for the database: the background process that reads entity changes and performs reindexing needs to read changed entities from the database.

Getting started

The `outbox-polling` coordination strategy requires an extra dependency. To implement this architecture, use the following Maven dependencies:

When integrating with Hibernate ORM

```
<dependency>
   <groupId>org.hibernate.search</groupId>
   <artifactId>hibernate-search-mapper-orm</artifactId>
   <version>8.4.0.CR1</version>
</dependency>
<dependency>
   <groupId>org.hibernate.search</groupId>
   <artifactId>hibernate-search-mapper-orm-outbox-polling</artifactId>
   <version>8.4.0.CR1</version>
</dependency>
<dependency>
   <groupId>org.hibernate.search</groupId>
   <artifactId>hibernate-search-backend-elasticsearch</artifactId>
   <version>8.4.0.CR1</version>
</dependency>
```

With the Standalone POJO Mapper (no Hibernate ORM)

> This architecture cannot be implemented with the Standalone POJO Mapper at the moment, because this mapper does not support coordination.

Also, configure coordination as explained in `outbox-polling`: additional event tables and polling in background processors.

Chapter 6. Hibernate ORM integration

6.1. Basics

The Hibernate ORM "mapper" is an integration of Hibernate Search into Hibernate ORM.

Its key features include:

- Listener-triggered indexing of Hibernate ORM entities as they are modified in the Hibernate ORM `EntityManager`/`Session`.
- Loading of managed entities as hits in the result of a search query.

6.2. Startup

The Hibernate Search integration into Hibernate ORM will start automatically, at the same time as Hibernate ORM, as soon as it is present in the classpath.

If for some reason you need to prevent Hibernate Search from starting, set the boolean property `hibernate.search.enabled` to `false`.

6.3. Shutdown

The Hibernate Search integration into Hibernate ORM will stop automatically, at the same time as Hibernate ORM.

On shutdown, Hibernate Search will stop accepting new indexing requests: new indexing attempts will throw exceptions. The Hibernate ORM shutdown will block until all ongoing indexing operations complete.

6.4. Mapping `Map`-based models

"Dynamic-map" entity models, i.e. models based on `java.util.Map` instead of custom classes, cannot be mapped using annotations. However, they can be mapped using the programmatic mapping API. You just need to refer to the types by their name using `context.programmaticMapping().type("thename")`:

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

6.5. Multi-tenancy with non-string tenant identifiers

While working with string tenant identifiers in Hibernate ORM has built-in support in Hibernate Search, using non-string tenant identifiers requires configuring a custom tenant identifier converter. This can be done by passing a bean reference of `TenantIdentifierConverter` type to the `hibernate.search.multi_tenancy.tenant_identifier_converter` configuration property.

6.6. Other configuration

Other configuration properties are mentioned in the relevant parts of this documentation. You can find a full reference of available properties in the ORM integration configuration properties appendix.

Chapter 7. Standalone POJO Mapper

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

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

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

7.1. Basics

The Standalone POJO Mapper enables mapping arbitrary POJOs to indexes.

Its key feature compared to the Hibernate ORM integration is its ability to run without Hibernate ORM or a relational database.

It can be used to index entities coming from an arbitrary datastore or even (though that's not recommended in general) to use Lucene or Elasticsearch as a primary datastore.

Because the Standalone POJO Mapper does not assume anything about the entities being mapped, beyond the fact they are represented as POJOs, it can be more complex to use than the Hibernate ORM integration. In particular:

- This mapper cannot detect entity changes on its own: all indexing must be explicit.
- Loading of entities as hits in the result of a search query must be implemented in the application.
- Loading of identifiers and entities for mass indexing must be implemented in the application.
- This mapper does not provide coordination between nodes at the moment.

7.2. Startup

Starting up Hibernate Search with the Standalone POJO Mapper is explicit and involves a builder:

Example 7.1: Starting up Hibernate Search with the Standalone POJO Mapper

```
CloseableSearchMapping searchMapping =
        SearchMapping.builder( AnnotatedTypeSource.fromClasses( ①
                Book.class, Associate.class, Manager.class
        ) )
                .property(
                        "hibernate.search.backend.hosts", ②
                        "elasticsearch.mycompany.com"
                )
                .build(); ③
```

① Create a builder, passing an `AnnotatedTypeSource` to let Hibernate Search know where to look for annotations.
② Set additional configuration properties (see also Configuration).
③ Build the `SearchMapping`.

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

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

7.3. Shutdown

You can shut down Hibernate Search with the Standalone POJO Mapper by calling the `close()` method on the mapping:

Example 7.2: Shutting down Hibernate Search with the Standalone POJO Mapper

```
CloseableSearchMapping searchMapping = /* ... */ ①
searchMapping.close(); ②
```

① Retrieve the `SearchMapping` that was returned when Hibernate Search started.
② Call `close()` to shut down Hibernate Search.

On shutdown, Hibernate Search will stop accepting new indexing requests: new indexing attempts will throw exceptions. The `close()` method will only return once all ongoing indexing operations complete.

7.4. Bean provider

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

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

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

The Standalone POJO Mapper can retrieve beans from CDI/Spring, but that support needs to be implemented explicitly through a bean provider.

You can plug in your own bean provider in two steps:

1. Define a class that implements the `org.hibernate.search.engine.environment.bean.spi.BeanProvider` interface.
2. Configure Hibernate Search to use that implementation by setting the configuration property `hibernate.search.bean_provider` to a bean reference pointing to the implementation, for example `class:com.mycompany.MyMappingConfigurer`. Obviously, the reference to the bean provider cannot be resolved using the bean provider.

7.5. Multi-tenancy

Multi-tenancy needs to be enabled explicitly when starting the Standalone POJO Mapper:

Example 7.3: Enabling multi-tenancy with the Standalone POJO Mapper

```
CloseableSearchMapping searchMapping = SearchMapping.builder( AnnotatedTypeSource.fromClasses( ①
        Book.class
) )
        // ...
        .property( "hibernate.search.mapping.multi_tenancy.enabled", true ) ②
        .build(); ③
```

① Create a builder.
② Enable multi-tenancy.
③ Build the `SearchMapping`.

Once multi-tenancy is enabled, a tenant ID will have to be provided when creating a `SearchSession` and in some other cases (creating a mass indexer, a workspace, …).

Example 7.4: Creating the `SearchSession` with a tenant identifier

```
SearchMapping searchMapping = /* ... */ ①
Object tenantId = "myTenantId";
try ( SearchSession searchSession = searchMapping.createSessionWithOptions() ②
        .tenantId( tenantId ) ③
        .build() ) { ④
    // ...
}
```

① Retrieve the `SearchMapping`.
② Start creating a new session.
③ Set the tenant identifier for the new session.
④ Build the new session.

When using non-string tenant identifiers, a custom `TenantIdentifierConverter` must be provided:

```
CloseableSearchMapping searchMapping = SearchMapping.builder( AnnotatedTypeSource.fromClasses( ①
        Book.class
```

```
    ) )
            // ...
            .property( "hibernate.search.mapping.multi_tenancy.enabled", true ) ②
            .property(
                    "hibernate.search.mapping.multi_tenancy.tenant_identifier_converter",
                    new TenantIdentifierConverter() {
                        @Override
                        public String toStringValue(Object tenantId) {
                            return tenantId == null ? null : tenantId.toString();
                        }

                        @Override
                        public Object fromStringValue(String tenantId) {
                            return tenantId == null ? null : UUID.fromString( tenantId );
                        }
                    }
            ) ③
            .build(); ④
```

① Create a builder.
② Enable multi-tenancy.
③ Set a custom tenant identifier converter.
④ Build the `SearchMapping`.

7.6. Mapping

While the Hibernate ORM integration can infer parts of the mapping from the Hibernate ORM mapping, the Standalone POJO Mapper cannot. As a result, the Standalone POJO Mapper needs more explicit configuration for its mapping:

- Entity types must be defined explicitly.
- Document identifiers must be mapped explicitly.
- The inverse side of associations must be mapped explicitly.

7.7. Indexing

7.7.1. Listener-triggered indexing

The Standalone POJO Mapper does not provide "implicit" indexing similar to the listener-triggered indexing in the Hibernate ORM integration.

Instead, you must index explicitly with an indexing plan.

7.7.2. Explicitly indexing on entity change events

The Standalone POJO Mapper can process entity change events (add, update, delete) and perform indexing accordingly, though events must necessarily be passed to Hibernate Search explicitly. See Indexing plans for more information about the API.

One major difference with the Hibernate ORM integration is that transactions (JTA or otherwise are not supported), so indexing is executed on session closing rather than on transaction commit.

7.7.3. Mass indexing

Because by default, the Standalone POJO Mapper does not know anything about where the entity data comes from, mass indexing requires plugging in a way to load entities *en masse* from the other datastore: a mass loading strategy.

Mass loading strategies are assigned to entity types as part of the entity definition: see Mass loading strategy for more information.

7.7.4. Entity loading in search queries

Because by default, the Standalone POJO Mapper does not know anything about where the entity data comes from, entity loading in search queries requires plugging in a way to load a selection of entities from the other datastore: a selection loading strategy.

Selection loading strategies are assigned to entity types as part of the entity definition: see Selection loading strategy for more information.

With the Standalone POJO Mapper, if you want entities to be loaded from the index, instead of an external datasource,

> add a projection constructor to your entity type.
>
> This will automatically result in your entity being loaded from the index when the configuration described in this section is missing and loading is required (for example when not using `select()` in a search query).

7.8. Coordination

The Standalone POJO Mapper does not provide any way to coordinate between nodes at the moment, so its behavior is roughly similar to that described in No coordination, except entity data extracting happens on session closing instead of happening on Hibernate ORM session flushes, and indexing happens immediately after that instead of happening on transaction commit.

7.9. Reading configuration properties from a file

The Standalone POJO Mapper `SearchMappingBuilder` can also take properties from a `Reader` compatible with `java.util.Properties#load(Reader)`:

Example 7.5: Loading configuration properties from a file using a `Reader`

```
try (
        Reader propertyFileReader = /* ... */ ①
) {
    CloseableSearchMapping searchMapping = SearchMapping.builder( AnnotatedTypeSource.empty() ) ②
            .properties( propertyFileReader ) ③
            .build();
}
```

① Get a reader representing a property file with configuration properties.
② Start configuring the Standalone POJO Mapper.
③ Pass the property file reader to the builder.

7.10. Other configuration

Other configuration properties are mentioned in the relevant parts of this documentation. You can find a full reference of available properties in the Standalone POJO Mapper configuration properties appendix.

Chapter 8. Configuration

8.1. Configuration sources

8.1.1. Configuration sources when integrating into Hibernate ORM

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

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

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

When setting properties through the configuration file of your framework, the keys of configuration properties will likely be different from the keys mentioned in this documentation.

For example `hibernate.search.backend.hosts` will become `quarkus.hibernate-search-orm.elasticsearch.hosts` in Quarkus or `spring.jpa.properties.hibernate.search.backend.hosts` in Spring Boot.

See Framework support for more information.

8.1.2. Configuration sources with the Standalone POJO mapper

When using Hibernate Search in the Standalone POJO Mapper (without Hibernate ORM), configuration properties must be set programmatically as you build the mapping.

See this section for more information.

8.2. Configuration properties

8.2.1. Structure of configuration properties

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

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

Global properties

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

Global properties are explained in the relevant parts of this documentation:

- Hibernate ORM integration
- Standalone POJO Mapper
- Mapping configuration
- And many more.

Backend properties

These properties affect a single backend. They are grouped under a common root:

- `hibernate.search.backend` for the default backend (most common usage).
- `hibernate.search.backends.<backend-name>` for a named backend (advanced usage).

Backend properties are explained in the relevant parts of this documentation:

- Lucene backend
- Elasticsearch backend

Index properties

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

With the root `hibernate.search.backend`, they set defaults for all indexes of the backend.

With the root `hibernate.search.backend.indexes.<index-name>`, they set the value for a specific index, overriding the defaults (if any). The backend and index names must match the names defined in the mapping. For Hibernate ORM entities, the default index

name is the name of the indexed class, without the package: `org.mycompany.Book` will have `Book` as its default index name. Index names can be customized in the mapping.

Alternatively, the backend can also be referenced by name, i.e. the roots above can also be `hibernate.search.backends.<backend-name>` or `hibernate.search.backends.<backend-name>.indexes.<index-name>`.

Examples:

- `hibernate.search.backend.io.commit_interval = 500` sets the `io.commit_interval` property for all indexes of the default backend.
- `hibernate.search.backend.indexes.Product.io.commit_interval = 2000` sets the `io.commit_interval` property for the `Product` index of the default backend.
- `hibernate.search.backends.myBackend.io.commit_interval = 500` sets the `io.commit_interval` property for all indexes of backend `myBackend`.
- `hibernate.search.backends.myBackend.indexes.Product.io.commit_interval = 2000` sets the `io.commit_interval` property for the `Product` index of backend `myBackend`.

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

- Lucene backend
- Elasticsearch backend

8.2.2. Building property keys programmatically

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

BackendSettings

`BackendSettings.backendKey(ElasticsearchBackendSettings.HOSTS)` is equivalent to `hibernate.search.backend.hosts`.

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

For a list of available property keys, see the Elasticsearch backend configuration properties appendix or the Lucene backend configuration properties appendix.

IndexSettings

`IndexSettings.indexKey("myIndex", ElasticsearchIndexSettings.INDEXING_QUEUE_SIZE)` is equivalent to `hibernate.search.backend.indexes.myIndex.indexing.queue_size`.

`IndexSettings.indexKey("myBackend", "myIndex", ElasticsearchIndexSettings.INDEXING_QUEUE_SIZE)` is equivalent to `hibernate.search.backends.myBackend.indexes.myIndex.indexing.queue_size`.

For a list of available property keys, see the Elasticsearch backend configuration properties appendix or the Lucene backend configuration properties appendix. Look for properties having a variant starting with a `hibernate.search.backend.indexes`.

Example 8.1: Using the helper to build hibernate configuration

```
private Properties buildHibernateConfiguration() {
    Properties config = new Properties();
    // backend configuration
    config.put( BackendSettings.backendKey( ElasticsearchBackendSettings.HOSTS ), "127.0.0.1:9200" );
    config.put( BackendSettings.backendKey( ElasticsearchBackendSettings.PROTOCOL ), "http" );
    // index configuration
    config.put(
            IndexSettings.indexKey( "myIndex", ElasticsearchIndexSettings.INDEXING_MAX_BULK_SIZE ),
            20
    );
    // orm configuration
    config.put(
            HibernateOrmMapperSettings.INDEXING_PLAN_SYNCHRONIZATION_STRATEGY,
            IndexingPlanSynchronizationStrategyNames.ASYNC
    );
    // engine configuration
    config.put( EngineSettings.BACKGROUND_FAILURE_HANDLER, "myFailureHandler" );
    return config;
}
```

8.2.3. Type of configuration properties

Property values can be set programmatically as Java objects, or through a configuration file as a string that will have to be parsed.

Each configuration property in Hibernate Search has an assigned type, and this type defines the accepted values in both cases.

Here are the definitions of all property types.

Designation	Accepted Java objects	Accepted String format
String	`java.lang.String`	Any string
Boolean	`java.lang.Boolean`	`true` or `false` (case-insensitive)
Integer	`java.lang.Number` (will call .intValue())	Any string that can be parsed by `Integer.parseInt`
Long	`java.lang.Number` (will call .longValue())	Any string that can be parsed by `Long.parseLong`
Bean reference of type T	An instance of `T` or `BeanReference` or a reference by type as a `java.lang.Class` (see Bean references)	See Parsing of bean references

When a configuration property of any type above is documented as multivalued, that property accepts either:

- A `java.util.Collection` containing any Java object that would be accepted for a single-valued property of the same type (see above);
- or a comma-separated string containing strings that would be accepted for a single-valued property of the same type (see above);
- or a single Java object that would be accepted for a single-valued property of the same type (see above).

8.3. Configuration property checking

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

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

8.4. Beans

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

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

8.4.1. Supported frameworks

Supported frameworks when integrating into Hibernate ORM

When using the Hibernate Search integration into Hibernate ORM, all dependency injection frameworks integrated into Hibernate ORM are automatically integrated into Hibernate Search.

This includes, but may not be limited to:

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

When not using a dependency injection framework, or when it is not integrated into Hibernate ORM, beans can only be retrieved using reflection by calling the public, no-arg constructor of the referenced type; see Bean resolution.

Supported frameworks when using the Standalone POJO Mapper

When using the Standalone POJO Mapper dependency injection support must be plugged in manually.

Failing that, beans can only be retrieved using reflection by calling the public, no-arg constructor of the referenced type; see Bean

resolution.

8.4.2. Bean references

Bean references are composed of two parts:

- The type, i.e. a `java.lang.Class`.
- Optionally, the name, as a `String`.

When referencing beans using a string value in configuration properties, the type is implicitly set to whatever interface Hibernate Search expects for that configuration property.

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

8.4.3. Parsing of bean references

When referencing beans using a string value in configuration properties, that string is parsed.

Here are the most common formats:

- `bean:` followed by the name of a Spring or CDI bean. For example `bean:myBean`.
- `class:` followed by the fully-qualified name of a class, to be instantiated through Spring/CDI if available, or through its public, no-argument constructor otherwise. For example `class:com.mycompany.MyClass`.
- An arbitrary string that doesn't contain a colon: it will be interpreted as explained in Bean resolution. In short:
  - first, look for a built-in bean with the given name;
  - then try to retrieve a bean with the given name from Spring/CDI (if available);
  - then try to interpret the string as a fully-qualified class name and to retrieve the corresponding bean from Spring/CDI (if available);
  - then try to interpret the string as a fully-qualified class name and to instantiate it through its public, no-argument constructor.

The following formats are also accepted, but are only useful for advanced use cases:

- `any:` followed by an arbitrary string. Equivalent to leaving out the prefix in most cases. Only useful if the arbitrary string contains a colon.
- `builtin:` followed by the name of a built-in bean, e.g. `simple` for the Elasticsearch index layout strategies. This will not fall back to Spring/CDI or a direct constructor call.
- `constructor:` followed by the fully-qualified name of a class, to be instantiated through its public, no-argument constructor. This will ignore built-in beans and will not try to instantiate the class through Spring/CDI.

8.4.4. Bean resolution

Bean resolution (i.e. the process of turning this reference into an object instance) happens as follows by default:

- If the given reference matches a built-in bean, that bean is used.

  Example: the name `simple`, when used as the value of the property `hibernate.search.backend.layout.strategy` to configure the Elasticsearch index layout strategy, resolves to the built-in `simple` strategy.

- Otherwise, if a supported Dependency Injection (DI) framework is available, the reference is resolved using the DI framework.
  - If a managed bean with the given type (and if provided, name) exists, that bean is used.

    Example: the name `myLayoutStrategy`, when used as the value of the property `hibernate.search.backend.layout.strategy` to configure the Elasticsearch index layout strategy, resolves to any bean known from CDI/Spring of type `IndexLayoutStrategy` and annotated with `@Named("myLayoutStrategy")`.

  - Otherwise, if a name is given, and that name is a fully-qualified class name, and a managed bean of that type exists, that bean is used.

    Example: the name `com.mycompany.MyLayoutStrategy`, when used as the value of the property `hibernate.search.backend.layout.strategy` to configure the Elasticsearch index layout strategy, resolves to any bean known from CDI/Spring and extending `com.mycompany.MyLayoutStrategy`.

- Otherwise, reflection is used to resolve the bean.
  - If a name is given, and that name is a fully-qualified class name, and that class extends the type reference, an instance is created by invoking the public, no-argument constructor of that class.

Example: the name `com.mycompany.MyLayoutStrategy`, when used as the value of the property `hibernate.search.backend.layout.strategy` to configure the Elasticsearch index layout strategy, resolves to an instance of `com.mycompany.MyLayoutStrategy`.

- If no name is given, an instance is created by invoking the public, no-argument constructor of the referenced type.

  Example: the class `com.mycompany.MyLayoutStrategy.class` (a `java.lang.Class`, not a `String`), when used as the value of the property `hibernate.search.backend.layout.strategy` to configure the Elasticsearch index layout strategy, resolves to an instance of `com.mycompany.MyLayoutStrategy`.

It is possible to control bean retrieval more finely by selecting a `BeanRetrieval`; see the javadoc of `org.hibernate.search.engine.environment.bean.BeanRetrieval` for more information. See also Parsing of bean references for the prefixes that allow to specify the bean retrieval when referencing a bean from configuration properties.

8.4.5. Bean injection

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

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

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

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

8.4.6. Bean lifecycle

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

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

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

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

Immutable beans or beans used only once such as `ElasticsearchAnalysisConfigurer` may be safely assigned any scope.

However, some beans are expected to be mutable and instantiated multiple times, such as for example `PropertyBinder`. For these beans, it is recommended to use the "dependent" scope (CDI terminology) or the "prototype" scope (Spring terminology). When in doubt, this is also generally the safest choice for beans injected into Hibernate Search.

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

8.5. Global configuration

This section presents global configuration, common to all mappers and backends.

8.5.1. Background failure handling

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

To customize background failure handling, you will need to:

1. Define a class that implements the `org.hibernate.search.engine.reporting.FailureHandler` interface.
2. Configure the backend to use that implementation by setting the configuration property `hibernate.search.background_failure_handler` to a bean reference pointing to the implementation, for example `class:com.mycompany.MyFailureHandler`.

Hibernate Search will call the `handle` methods whenever a failure occurs.

Example 8.2: Implementing and using a `FailureHandler`

```
package org.hibernate.search.documentation.reporting.failurehandler;

import org.hibernate.search.engine.common.EntityReference;
import org.hibernate.search.engine.reporting.EntityIndexingFailureContext;
import org.hibernate.search.engine.reporting.FailureContext;
import org.hibernate.search.engine.reporting.FailureHandler;
import org.hibernate.search.util.impl.test.extension.StaticCounters;

public class MyFailureHandler implements FailureHandler {

    @Override
    public void handle(FailureContext context) { ①
        String failingOperationDescription = context.failingOperation().toString(); ②
        Throwable throwable = context.throwable(); ③

        // ... report the failure ... ④
    }

    @Override
    public void handle(EntityIndexingFailureContext context) { ⑤
        String failingOperationDescription = context.failingOperation().toString();
        Throwable throwable = context.throwable();
        for ( EntityReference entityReference : context.failingEntityReferences() ) { ⑥
            Class<?> entityType = entityReference.type(); ⑦
            String entityName = entityReference.name(); ⑦
            Object entityId = entityReference.id(); ⑦
            String entityReferenceAsString = entityReference.toString(); ⑧

            // ... process the entity reference ... ⑨
        }
    }

}
```

① `handle(FailureContext)` is called for generic failures that do not fit any other specialized `handle` method.
② Get a description of the failing operation from the context.
③ Get the throwable thrown when the operation failed from the context.
④ Use the context-provided information to report the failure in any relevant way.
⑤ `handle(EntityIndexingFailureContext)` is called for failures occurring when indexing entities.
⑥ On top of the failing operation and throwable, the context also lists references to entities that could not be indexed correctly because of the failure.
⑦ Entity references expose the entity type, name and identifier.
⑧ Entity references may be converted to a human-readable string using `toString()`.
⑨ Use the context-provided information to report the failure in any relevant way.

```
hibernate.search.background_failure_handler =
class:org.hibernate.search.documentation.reporting.failurehandler.MyFailureHandler
```

Assign the background failure handler using a Hibernate Search configuration property.

When a failure handler's `handle` method throws an error or exception, Hibernate Search will catch it and log it at the ERROR level. It will not be propagated.

8.5.2. Multi-tenancy

If your application uses Hibernate ORM's multi-tenancy support, or if you configured multi-tenancy explicitly in the Standalone POJO Mapper, Hibernate Search should detect that and configure your backends transparently. For details, see:

- here for Lucene
- here for Elasticsearch

In some cases, in particular when using the `outbox-polling` coordination strategy or when expecting the mass indexer to implicitly target

all tenants, you will need to list explicitly all tenant identifiers that your application might use. This information is used by Hibernate Search when spawning background processes that should apply an operation to every tenant.

The list of identifiers is defined through the following configuration property:

```
hibernate.search.multi_tenancy.tenant_ids = mytenant1,mytenant2,mytenant3
```

This property may be set to a String containing multiple tenant identifiers separated by commas, or a `Collection<String>` containing tenant identifiers.

Chapter 9. Main API Entry Points

This section details the main entry points to Hibernate Search APIs at runtime, i.e. APIs to index, search, look up metadata, …

9.1. SearchMapping

9.1.1. Basics

The SearchMapping is the top-most entrypoint to Hibernate Search APIs: it represents the whole mapping from entities to indexes.

The SearchMapping is thread-safe: it can safely be used concurrently from multiple threads. However, that does not mean the objects it returns (SearchWorkspace, …) are themselves thread-safe.

The SearchMapping in Hibernate Search is the equivalent of the EntityManagerFactory/SessionFactory in JPA/Hibernate ORM.

Some frameworks, such as Quarkus, allow you to simply @Inject the SearchMapping into your CDI beans.

9.1.2. Retrieving the SearchMapping with the Hibernate ORM integration

With the Hibernate ORM integration, the SearchMapping is created automatically when Hibernate ORM starts.

To retrieve the SearchMapping, call Search.mapping(…) and pass the EntityManagerFactory/SessionFactory:

Example 9.1: Retrieving the SearchMapping from a Hibernate ORM SessionFactory

```
SessionFactory sessionFactory = /* ... */ ①
SearchMapping searchMapping = Search.mapping( sessionFactory ); ②
```

① Retrieve the SessionFactory. Details depend on your framework, but this is generally achieved by injecting it into your own class, e.g. by annotating a field of that type with @Inject or @PersistenceUnit.

② Call Search.mapping(…), passing the SessionFactory as an argument. This will return the SearchMapping.

Still with the Hibernate ORM integration, the same can be done from a JPA EntityManagerFactory:

Example 9.2: Retrieving the SearchMapping from a JPA EntityManagerFactory

```
EntityManagerFactory entityManagerFactory = /* ... */ ①
SearchMapping searchMapping = Search.mapping( entityManagerFactory ); ②
```

① Retrieve the EntityManagerFactory. Details depend on your framework, but this is generally achieved by injecting it into your own class, e.g. by annotating a field of that type with @Inject or @PersistenceUnit.

② Call Search.mapping(…), passing the EntityManagerFactory as an argument. This will return the SearchMapping.

9.1.3. Retrieving the SearchMapping with the Standalone POJO Mapper

With the Standalone POJO Mapper, the SearchMapping is the result of starting Hibernate Search.

See this section for more information about starting Hibernate Search with the Standalone POJO Mapper.

9.2. SearchSession

9.2.1. Basics

The SearchSession represents the context in which a sequence of related operations are executed. It should generally be used for a very short time, for example to process a single web request.

The SearchSession is **not** thread-safe: it must not be used concurrently from multiple threads.

The SearchSession in Hibernate Search is the equivalent of the EntityManager/Session in JPA/Hibernate ORM.

Some frameworks, such as Quarkus, allow you to simply `@Inject` the `SearchSession` into your CDI beans.

9.2.2. Retrieving the `SearchSession` with the Hibernate ORM integration

To retrieve the `SearchSession` with the Hibernate ORM integration, call `Search.session(…)` and pass the `EntityManager`/`Session`:

Example 9.3: Retrieving the `SearchSession` from a Hibernate ORM `Session`

```
Session session = /* ... */ ①
SearchSession searchSession = Search.session( session ); ②
```

① Retrieve the `Session`. Details depend on your framework, but this is generally achieved by injecting it into your own class, e.g. by annotating a field of that type with `@Inject` or `@PersistenceContext`.

② Call `Search.session(…)`, passing the `Session` as an argument. This will return the `SearchSession`.

Still with the Hibernate ORM integration, the same can be done from a JPA `EntityManager`:

Example 9.4: Retrieving the `SearchSession` from a JPA `EntityManager`

```
EntityManager entityManager = /* ... */ ①
SearchSession searchSession = Search.session( entityManager ); ②
```

① Retrieve the `EntityManager`. Details depend on your framework, but this is generally achieved by injecting it into your own class, e.g. by annotating a field of that type with `@Inject` or `@PersistenceContext`.

② Call `Search.mapping(…)`, passing the `EntityManager` as an argument. This will return the `SearchSession`.

9.2.3. Retrieving the `SearchSession` with the Standalone POJO Mapper

With the Standalone POJO Mapper, the `SearchSession` should be created and closed explicitly:

Example 9.5: Creating the `SearchSession`

```
SearchMapping searchMapping = /* ... */ ①
try ( SearchSession searchSession = searchMapping.createSession() ) { ②
    // ...
}
```

① Retrieve the `SearchMapping`.

② Create a new session. Note we're using a try-with-resources block, so that the session will automatically be closed when we're done with it, which will in particular trigger the execution of the indexing plan.

Forgetting to close the `SearchSession` will lead to indexing not being executed, and may even cause memory leaks.

The `SearchSession` can also be configured with a few options:

Example 9.6: Creating the `SearchSession` with options

```
SearchMapping searchMapping = /* ... */ ①
Object tenantId = "myTenant";
try ( SearchSession searchSession = searchMapping.createSessionWithOptions() ②
        .indexingPlanSynchronizationStrategy( IndexingPlanSynchronizationStrategy.sync() )③
        .tenantId( tenantId )
        .build() ) { ④
    // ...
}
```

① Retrieve the `SearchMapping`.

② Start creating a new session. Note we're using a try-with-resources block, so that the session will automatically be closed when we're done with it, which will in particular trigger the execution of the indexing plan.

③ Pass options to the new session.
④ Build the new session.

9.3. SearchScope

The `SearchScope` represents a set of indexed entities and their indexes.

The `SearchScope` is thread-safe: it can safely be used concurrently from multiple threads. However, that does not mean the objects it returns (`SearchWorkspace`, …) are themselves thread-safe.

A `SearchScope` can be retrieved from a `SearchMapping` as well as from a `SearchSession`.

Example 9.7: Retrieving a `SearchScope` from a `SearchMapping`

```
SearchMapping searchMapping = /* ... */ ①
SearchScope<Book> bookScope = searchMapping.scope( Book.class ); ②
SearchScope<Person> associateAndManagerScope =
        searchMapping.scope( Arrays.asList( Associate.class, Manager.class ) ); ③
SearchScope<Person> personScope = searchMapping.scope( Person.class ); ④
SearchScope<Person> personSubTypesScope = searchMapping.scope( Person.class,
        Arrays.asList( "Manager", "Associate" ) ); ⑤
SearchScope<Object> allScope = searchMapping.scope( Object.class ); ⑥
```

① Retrieve the `SearchMapping`.
② Create a `SearchScope` targeting the `Book` entity type only.
③ Create a `SearchScope` targeting both the `Associate` entity type and the `Manager` entity type. The scope's generic type parameter can be any common supertype of those entity types.
④ A scope will always target all subtypes of the given classes, and the given classes do not need to be indexed entity types themselves. This creates a `SearchScope` targeting all (indexed entity) subtypes of the `Person` interface; in our case this will target both the `Associate` entity type and the `Manager` entity type.
⑤ For advanced use cases, it is possible to target entity types by their name. For Hibernate ORM this would be the JPA entity name, and for the Standalone POJO Mapper this would be the name assigned to the entity type upon entity definition. In both cases, the entity name is the simple name of the Java class by default.
⑥ Passing `Object.class` will create a scope targeting every single indexed entity types.

Example 9.8: Retrieving a `SearchScope` from a `SearchSession`

```
SearchSession searchSession = /* ... */ ①
SearchScope<Book> bookScope = searchSession.scope( Book.class ); ②
SearchScope<Person> associateAndManagerScope =
        searchSession.scope( Arrays.asList( Associate.class, Manager.class ) ); ③
SearchScope<Person> personScope = searchSession.scope( Person.class ); ④
SearchScope<Person> personSubTypesScope = searchSession.scope( Person.class,
        Arrays.asList( "Manager", "Associate" ) ); ⑤
SearchScope<Object> allScope = searchSession.scope( Object.class ); ⑥
```

① Retrieve the `SearchSession`.
② Create a `SearchScope` targeting the `Book` entity type only.
③ Create a `SearchScope` targeting both the `Associate` entity type and the `Manager` entity type. The scope's generic type parameter can be any common supertype of those entity types.
④ A scope will always target all subtypes of the given classes, and the given classes do not need to be indexed entity types themselves. This creates a `SearchScope` targeting all (indexed entity) subtypes of the `Person` interface; in our case this will target both the `Associate` entity type and the `Manager` entity type.
⑤ For advanced use cases, it is possible to target entity types by their name. For Hibernate ORM this would be the JPA entity name, and for the Standalone POJO Mapper this would be the name assigned to the entity type upon entity definition. In both cases, the entity name is the simple name of the Java class by default.
⑥ Passing `Object.class` will create a scope targeting every single indexed entity types.

Chapter 10. Mapping entities to indexes

10.1. Configuring the mapping

10.1.1. Annotation-based mapping

The main way to map entities to indexes is through annotations, as explained in Entity definition, Entity/index mapping and the following sections.

By default, Hibernate Search will automatically process mapping annotations for entity types, as well as nested types in those entity types, for instance embedded types.

Annotation-based mapping can be disabled by setting `hibernate.search.mapping.process_annotations` to `false` for the Hibernate ORM integration, or through `AnnotationMappingConfigurationContext` for any mapper: see Mapping configurer to access that context, and see the javadoc of `AnnotationMappingConfigurationContext` for available options.

If you disable annotation-based mapping, you will probably need to configure the mapping programmatically: see Programmatic mapping.

Hibernate Search will also try to find some annotated types through classpath scanning.

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

10.1.2. Classpath scanning

Basics

Hibernate Search will automatically scan the JARs of entity types on startup, looking for types annotated with "root mapping annotations" so that it can automatically add those types to the list of types whose annotations should be processed.

Root mapping annotations are mapping annotations that serve as the entrypoint to a mapping, for example `@ProjectionConstructor` or custom root mapping annotations. Without this scanning, Hibernate Search would learn about e.g. projection constructors too late (when the projection is actually executed) and would fail due to a lack of metadata.

The scanning is backed by Jandex, a library that indexes the content of JARs.

Scanning dependencies of the application

By default, Hibernate Search will only scan the JARs containing your Hibernate ORM entities.

If you want Hibernate Search to detect types annotated with root mapping annotations in other JARs, you will first need to access an `AnnotationMappingConfigurationContext`.

From that context, either:

- call `annotationMappingContext.add( MyType.class )` to explicitly tell Hibernate Search to process annotation on `MyType`, and to discover other types annotated with root mapping annotations in the JAR containing `MyType`.
- OR (advanced usage, incubating) call `annotationMappingContext.addJandexIndex( <an IndexView instance> )` to explicitly tell Hibernate Search to look for types annotated with root mapping annotations in the given Jandex index.

Configuring scanning

Hibernate Search's scanning may trigger the indexing of JARs through Jandex on application startup. In some of the more complicated environments, this indexing may not be able to get access to classes to index, or may unnecessarily slow down startup.

Running Hibernate Search within Quarkus or Wildfly has its benefits as:

- With the Quarkus framework, scanning part of the Hibernate Search's startup is executed at build time and the indexes are provided to it automatically.
- With the WildFly application server, this part of Hibernate Search's startup is executed in an optimized way and the indexes are provided to it automatically as well.

In other cases, depending on the application needs, the Jandex Maven Plugin can be used during the building stage of the application, so that indexes are already built and ready when the application starts.

Alternatively, If your application does not use `@ProjectionConstructor` or custom root mapping annotations, you may want to disable this feature entirely or partially.

This is not recommended in general as it may lead to bootstrap failures or ignored mapping annotations because Hibernate Search will no longer be able to automatically discover types annotated with root annotations in JARs that do not have an embedded Jandex index.

Two options are available for this:

- Setting `hibernate.search.mapping.discover_annotated_types_from_root_mapping_annotations` to `false` will disable any attempts of automatic discovery, even if there is a Jandex index available, partial or full, which may help if there are no types annotated with root mapping annotations at all, or if they are listed explicitly through a mapping configurer or through an `AnnotatedTypeSource`.
- Setting `hibernate.search.mapping.build_missing_discovered_jandex_indexes` to `false` will disable Jandex index building on startup, but will still use any pre-built Jandex indexes available. This may help if partial automatic discovery is required, i.e. available indexes will be used for discovery, but sources that do not have an index available will be ignored unless their `@ProjectionConstructor` -annotated types are listed explicitly through a mapping configurer or through an `AnnotatedTypeSource`.

10.1.3. Programmatic mapping

Most examples in this documentation use annotation-based mapping, which is generally enough for most applications. However, some applications have needs that go beyond what annotations can offer:

- a single entity type must be mapped differently for different deployments — e.g. for different customers.
- many entity types must be mapped similarly, without code duplication.

To address those needs, you can use *programmatic* mapping: define the mapping through code that will get executed on startup.

Programmatic mapping is configured through `ProgrammaticMappingConfigurationContext`: see Mapping configurer to access that context.

By default, programmatic mapping will be merged with annotation mapping (if any).

To disable annotation mapping, see Annotation-based mapping.

Programmatic mapping is declarative and exposes the exact same features as annotation-based mapping.

In order to implement more complex, "imperative" mapping, for example to combine two entity properties into a single index field, use custom bridges.

Alternatively, if you only need to repeat the same mapping for several types or properties, you can apply a custom annotation on those types or properties, and have Hibernate Search execute some programmatic mapping code when it encounters that annotation. This solution doesn't require mapper-specific configuration.

See Custom mapping annotations for more information.

10.1.4. Mapping configurer

Hibernate ORM integration

With the Hibernate ORM integration, a custom `HibernateOrmSearchMappingConfigurer` can be plugged into Hibernate Search in order to configure annotation mapping (`AnnotationMappingConfigurationContext`), programmatic mapping (`ProgrammaticMappingConfigurationContext`), and more.

Plugging in a custom configurer requires two steps:

1. Define a class that implements the `org.hibernate.search.mapper.orm.mapping.HibernateOrmSearchMappingConfigurer` interface.
2. Configure Hibernate Search to use that implementation by setting the configuration property `hibernate.search.mapping.configurer` to a bean reference pointing to the implementation, for example `class:com.mycompany.MyMappingConfigurer`.

You can pass multiple bean references separated by commas. See Type of configuration properties.

Hibernate Search will call the `configure` method of this implementation on startup, and the configurer will be able to take advantage of a DSL to configure annotation mapping or define the programmatic mapping, for example:

Example 10.1: Implementing a mapping configurer with the Hibernate ORM integration

```
public class MySearchMappingConfigurer implements HibernateOrmSearchMappingConfigurer {
    @Override
    public void configure(HibernateOrmMappingConfigurationContext context) {
        ProgrammaticMappingConfigurationContext mapping = context.programmaticMapping(); ①
        TypeMappingStep bookMapping = mapping.type( Book.class ); ②
        bookMapping.indexed(); ③
        bookMapping.property( "title" ) ④
                .fullTextField().analyzer( "english" ); ⑤
    }
}
```

① Access the programmatic mapping.

② Access the programmatic mapping of type `Book`.

③ Define `Book` as indexed.

④ Access the programmatic mapping of property `title` of type `Book`.

⑤ Define an index field based on property `title` of type `Book`.

Standalone POJO Mapper

The Standalone POJO Mapper does not offer a "mapping configurer" at the moment (HSEARCH-4615). However, `AnnotationMappingConfigurationContext` and `ProgrammaticMappingConfigurationContext` can be accessed when building the `SearchMapping`:

With the Hibernate ORM integration, a custom `StandalonePojoMappingConfigurer` can be plugged into Hibernate Search in order to configure annotation mapping (`AnnotationMappingConfigurationContext`), programmatic mapping (`ProgrammaticMappingConfigurationContext`), and more.

Plugging in a custom configurer requires two steps:

1. Define a class that implements the `org.hibernate.search.mapper.pojo.standalone.mapping.StandalonePojoMappingConfigurer` interface.
2. Configure Hibernate Search to use that implementation by setting the configuration property `hibernate.search.mapping.configurer` to a bean reference pointing to the implementation, for example `class:com.mycompany.MyMappingConfigurer`.

You can pass multiple bean references separated by commas. See Type of configuration properties.

Hibernate Search will call the `configure` method of this implementation on startup, and the configurer will be able to take advantage of a DSL to configure annotation mapping or define the programmatic mapping, for example:

Example 10.2: Implementing a mapping configurer with the Standalone POJO Mapper

```
public class MySearchMappingConfigurer implements StandalonePojoMappingConfigurer {
    @Override
    public void configure(StandalonePojoMappingConfigurationContext context) {
        context.annotationMapping() ①
                .discoverAnnotationsFromReferencedTypes( false )
                .discoverAnnotatedTypesFromRootMappingAnnotations( false );

        ProgrammaticMappingConfigurationContext mappingContext = context.programmaticMapping(); ②
        TypeMappingStep bookMapping = mappingContext.type( Book.class ); ③
        bookMapping.searchEntity(); ④
        bookMapping.indexed(); ⑤
        bookMapping.property( "id" ) ⑥
                .documentId(); ⑦
        bookMapping.property( "title" ) ⑧
                .fullTextField().analyzer( "english" ); ⑨
    }
}
```

① Access the annotation mapping context to configure annotation mapping.

② Access the programmatic mapping context to configure programmatic mapping.

③ Access the programmatic mapping of type `Book`.

④ Define `Book` as an entity type.
⑤ Define `Book` as indexed.
⑥ Access the programmatic mapping of property `id` of type `Book`.
⑦ Define the identifier of type `Book` as its property `id`.
⑧ Access the programmatic mapping of property `title` of type `Book`.
⑨ Define an index field based on property `title` of type `Book`.

10.2. Entity definition

10.2.1. Basics

Before a type can be mapped to indexes, Hibernate Search needs to be aware of which types in the application domain model are entity types.

When indexing Hibernate ORM entities, the entity types are fully defined by Hibernate ORM (generally through Jakarta's `@Entity` annotation), and no explicit definition is necessary: **you can safely skip this entire section**.

When using the Standalone POJO Mapper, entity types need to be defined explicitly.

10.2.2. Explicit entity definition

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

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

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

`@SearchEntity` and its corresponding programmatic mapping `.searchEntity()` are unnecessary for Hibernate ORM entities, and in fact unsupported when using the Hibernate ORM integration.

See HSEARCH-5076 to track progress on allowing the use of `@SearchEntity` in the Hibernate ORM integration to map non-ORM entities.

With the Standalone POJO Mapper, entity types must be marked explicitly with the `@SearchEntity` annotation.

Example 10.3: Marking a class as an entity with `@SearchEntity`

```
@SearchEntity ①
@Indexed ②
public class Book {
```

① Annotate the type with `@SearchEntity`
② `@Indexed` is optional: it is only necessary if you intend to map this type to an index.

Not all types are entity types, even if they have a composite structure.

Incorrectly marking types as entity types may force you to add unnecessary complexity to your domain model, such as defining identifiers or an inverse side for "associations" to such types that won't get used.

Make sure to read this section for more information on what entity types are and why they are necessary.

Subclasses do **not** inherit the `@SearchEntity` annotation.

Each subclass must be annotated with `@SearchEntity` as well, or it will not be considered as an entity by Hibernate Search.

However, for subclasses that are also annotated with `@SearchEntity`, some entity-related configuration can be inherited; see the relevant sections for details.

By default, with the Standalone POJO Mapper:

- The entity name will be equal to the class' simple name (`java.lang.Class#getSimpleName`).
- The entity will not be configured for loading, be it to return entities as hits in search queries or for mass indexing.

See the following sections to override these defaults.

10.2.3. Entity name

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

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

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

The entity name, distinct from the name of the corresponding class, is involved in various places, including but not limited to:

- as the default index name for `@Indexed`;
- for entity type name mapping in Elasticsearch;
- to target entity types using a String.

The entity name defaults to the class' simple name (`java.lang.Class#getSimpleName`).

Changing the entity name of an indexed entity may require full reindexing, in particular when using the Elasticsearch/OpenSearch backend.

See this section for more information.

With the Hibernate ORM integration, this name may be overridden through various means, but usually is through Jakarta Persistence's `@Entity` annotation, i.e. with `@Entity(name = …)`.

With the Standalone POJO Mapper, entity types are defined with `@SearchEntity`, and the entity name may be overridden with `@SearchEntity(name = …)`.

`@SearchEntity` and its corresponding programmatic mapping `.searchEntity()` are unnecessary for Hibernate ORM entities, and in fact unsupported when using the Hibernate ORM integration.

See HSEARCH-5076 to track progress on allowing the use of `@SearchEntity` in the Hibernate ORM integration to map non-ORM entities.

Example 10.4: Setting a custom entity name with `@SearchEntity(name = …)`

```
@SearchEntity(name = "MyAuthorName")
@Indexed
public class Author {
```

10.2.4. Mass loading strategy

A "mass loading strategy" gives Hibernate Search the ability to load entities of a given type for mass indexing.

With the Hibernate ORM integration, a mass loading strategy gets configured automatically for every single Hibernate ORM entity, and no further configuration is required.

With the Standalone POJO Mapper, entity types are defined with `@SearchEntity`, and, in order to take advantage of mass indexing, a mass loading strategy must be applied explicitly with `@SearchEntity(loadingBinder = …)`.

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

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

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

`@SearchEntity` and its corresponding programmatic mapping `.searchEntity()` are unnecessary for Hibernate ORM entities, and in fact unsupported when using the Hibernate ORM integration.

See HSEARCH-5076 to track progress on allowing the use of `@SearchEntity` in the Hibernate ORM integration to map non-ORM entities.

Example 10.5: Assigning a mass loading strategy with the Standalone POJO Mapper

```
@SearchEntity(loadingBinder = @EntityLoadingBinderRef(type = MyLoadingBinder.class)) ①
@Indexed
public class Book {
```

① Assign a loading binder to the entity.

Subclasses inherit the loading binder of their parent class, unless they override it with a loading binder of their own.

```
@Singleton
public class MyLoadingBinder implements EntityLoadingBinder { ①
    private final MyDatastore datastore;

    @Inject ②
    public MyLoadingBinder(MyDatastore datastore) {
        this.datastore = datastore;
    }

    @Override
    public void bind(EntityLoadingBindingContext context) { ③
        context.massLoadingStrategy( ④
                Book.class, ⑤
                new MyMassLoadingStrategy<>( datastore, Book.class ) ⑥
        );
    }
}
```

① The binder must implement the `EntityLoadingBinder` interface.
② Inject the implementation-specific datastore into the loading binder, for example here using CDI (or `@Autowired` on Spring, or …).
③ Implement the `bind` method.
④ Call `context.massLoadingStrategy(…)` to define the loading strategy to use.
⑤ Pass the expected supertype of loaded entities.
⑥ Pass the loading strategy.

Using injection in the loading binder with the Standalone POJO Mapper requires providing a `BeanProvider` through additional configuration.

Below is an example of `MassLoadingStrategy` implementation for an imaginary datastore.

Example 10.6: Implementing `MassLoadingStrategy`

```
public class MyMassLoadingStrategy<E>
        implements MassLoadingStrategy<E, String> {

    private final MyDatastore datastore; ①
    private final Class<E> rootEntityType;

    public MyMassLoadingStrategy(MyDatastore datastore, Class<E> rootEntityType) {
        this.datastore = datastore;
        this.rootEntityType = rootEntityType;
    }

    @Override
    public MassIdentifierLoader createIdentifierLoader(
            LoadingTypeGroup<E> includedTypes, ②
            MassIdentifierSink<String> sink, MassLoadingOptions options) {
```

```
            int batchSize = options.batchSize(); ③
            Collection<Class<? extends E>> typeFilter =
                    includedTypes.includedTypesMap().values(); ④
            return new MassIdentifierLoader() {
                private final MyDatastoreConnection connection =
                        datastore.connect(); ⑤
                private final MyDatastoreCursor<String> identifierCursor =
                        connection.scrollIdentifiers( typeFilter );

                @Override
                public void close() {
                    connection.close(); ⑤
                }

                @Override
                public OptionalLong totalCount() { ⑥
                    return OptionalLong.of( connection.countEntities( typeFilter ) );
                }

                @Override
                public void loadNext() throws InterruptedException {
                    List<String> batch = identifierCursor.next( batchSize );
                    if ( batch != null ) {
                        sink.accept( batch ); ⑦
                    }
                    else {
                        sink.complete(); ⑧
                    }
                }
            };
        }

        @Override
        public MassEntityLoader<String> createEntityLoader(
                LoadingTypeGroup<E> includedTypes, ⑨
                MassEntitySink<E> sink, MassLoadingOptions options) {
            return new MassEntityLoader<String>() {
                private final MyDatastoreConnection connection =
                        datastore.connect(); ⑩

                @Override
                public void close() { ⑧
                    connection.close();
                }

                @Override
                public void load(List<String> identifiers)
                        throws InterruptedException {
                    sink.accept( ⑪
                            connection.loadEntitiesById( rootEntityType, identifiers )
                    );
                }
            };
        }
    }
```

① The strategy must have access to the datastore to be able to open connections, but it should not generally have any open connection.

② Implement an identifier loader to retrieve the identifiers of all entities that will have to be indexed. Hibernate Search will only call this method once per mass indexing.

③ Retrieve the batch size configured on the `MassIndexer`. This defines how many IDs (at most) must be returned in each `List` passed to the sink.

④ Retrieve the list of entity types to be loaded: Hibernate Search may request loading of multiple types from a single loader if those types share similar mass loading strategies (see tips/warnings below).

⑤ The identifier loader owns a connection exclusively: it should create one when it's created, and close it when it's closed. Related: the identifier loader always executes in the same thread.

⑥ Count the number of entities to index. This is just an estimate: it can be off to some extent, but that will lead to incorrect reporting in the monitor (by default, the logs).

⑦ Retrieve identifiers in successive batches, one per call to `loadNext()`, and pass them to the sink.
⑧ When there are no more identifiers to load, let the sink know by calling `complete()`.
⑨ Implement an entity loader to actually load entities from the identifiers retrieved above. Hibernate Search will call this method multiple times for a single mass indexing, to create multiple loaders that execute in parallel.
⑩ Each entity loader owns a connection exclusively: it should create one when it's created, and close it when it's closed. Related: each entity loader always executes in the same thread.
⑪ Load the entities corresponding to the identifiers passed in argument and pass them to the sink. Entities passed to the sink do not need to be in the same order as the identifiers passed in argument.

Hibernate Search will optimize loading by grouping together types that have the same `MassLoadingStrategy`, or different strategies that are equal according to `equals()`/`hashCode()`.

When grouping types together, only one of the strategies will be called, and it will get passed a "type group" that includes all types that should be loaded.

This happens in particular when configuring the loading binder from a "parent" entity type is inherited by subtypes, and sets the same strategy on subtypes.

Be careful of non-abstract (instantiable) parent classes in inheritance trees: when the "type group" passed to the `createIdentifierLoader` method contains a parent type (say, `Animal`) and none of the subtypes (neither `Lion` nor `Zebra`), then the loader really should only load identifiers of instances of the parent type, not of its subtypes (it should load identifiers of entities whose type is exactly `Animal`, not `Lion` nor `Zebra`).

Once all types to reindex have their mass loading strategy implemented and assigned, they can be reindexed using the mass indexer:

Example 10.7: Mass indexing with the Standalone POJO Mapper

```
SearchMapping searchMapping = /* ... */ ①
searchMapping.scope( Object.class ).massIndexer() ②
        .startAndWait(); ③
```

① Retrieve the `SearchMapping`.
② Create a `MassIndexer` targeting every indexed entity type.
③ Start the mass indexing process and return when it is over.

10.2.5. Selection loading strategy

A "selection loading strategy" gives Hibernate Search the ability to load entities of a given type to return entities loaded from an external source as hits in search queries.

With the Hibernate ORM integration, a selection loading strategy gets configured automatically for every single Hibernate ORM entity, and no further configuration is required.

With the Standalone POJO Mapper, entity types are defined with `@SearchEntity`, and, in order to return entities loaded from an external source in search queries, a selection loading strategy must be applied explicitly with `@SearchEntity(loadingBinder = …)`.

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

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

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

`@SearchEntity` and its corresponding programmatic mapping `.searchEntity()` are unnecessary for Hibernate ORM entities, and in fact unsupported when using the Hibernate ORM integration.

See HSEARCH-5076 to track progress on allowing the use of `@SearchEntity` in the Hibernate ORM integration to map non-ORM entities.

Example 10.8: Assigning a selection loading strategy with the Standalone POJO Mapper

```
@SearchEntity(loadingBinder = @EntityLoadingBinderRef(type = MyLoadingBinder.class)) ①
@Indexed
public class Book {
```

① Assign a loading binder to the entity.

Subclasses inherit the loading binder of their parent class, unless they override it with a loading binder of their own.

```
@Singleton
public class MyLoadingBinder implements EntityLoadingBinder { ①
    @Override
    public void bind(EntityLoadingBindingContext context) { ②
        context.selectionLoadingStrategy( ③
                Book.class, ④
                new MySelectionLoadingStrategy<>( Book.class ) ⑤
        );
    }
}
```

① The binder must implement the `EntityLoadingBinder` interface.
② Implement the `bind` method.
③ Call `context.selectionLoadingStrategy(…)` to define the loading strategy to use.
④ Pass the expected supertype of loaded entities.
⑤ Pass the loading strategy.

Below is an example of `SelectionLoadingStrategy` implementation for an imaginary datastore.

Example 10.9: Implementing `SelectionLoadingStrategy`

```
public class MySelectionLoadingStrategy<E>
        implements SelectionLoadingStrategy<E> {
    private final Class<E> rootEntityType;

    public MySelectionLoadingStrategy(Class<E> rootEntityType) {
        this.rootEntityType = rootEntityType;
    }

    @Override
    public SelectionEntityLoader<E> createEntityLoader(
            LoadingTypeGroup<E> includedTypes, ①
            SelectionLoadingOptions options) {
        MyDatastoreConnection connection =
                options.context( MyDatastoreConnection.class ); ②
        return new SelectionEntityLoader<E>() {
            @Override
            public List<E> load(List<?> identifiers, Deadline deadline) {
                return connection.loadEntitiesByIdInSameOrder( ③
                        rootEntityType, identifiers );
            }
        };
    }
}
```

① Implement an entity loader to actually load entities from the identifiers returned by Lucene/Elasticsearch. Hibernate Search will call this method multiple times for a single mass indexing,
② The entity loader does not own a connection, but retrieves it from the context passed to the `SearchSession` (see next example).
③ Load the entities corresponding to the identifiers passed in argument and return them. Returned entities **must** be in the same order as the identifiers passed in argument.

Hibernate Search will optimize loading by grouping together types that have the same `SelectionLoadingStrategy`, or

different strategies that are equal according to `equals()`/`hashCode()`.

When grouping types together, only one of the strategies will be called, and it will get passed a "type group" that includes all types that should be loaded.

This happens in particular when configuring the loading binder from a "parent" entity type is inherited by subtypes, and sets the same strategy on subtypes.

Once all types to search for have their selection loading strategy implemented and assigned, they can be loaded as hits when querying:

Example 10.10: Loading entities as search query hits with the Standalone POJO Mapper

```
MyDatastore datastore = /* ... */ ①
SearchMapping searchMapping = /* ... */ ②
try ( MyDatastoreConnection connection = datastore.connect(); ③
        SearchSession searchSession = searchMapping.createSessionWithOptions() ④
                .loading( o -> o.context( MyDatastoreConnection.class, connection ) ) ⑤
                .build() ) { ⑥
    List<Book> hits = searchSession.search( Book.class ) ⑦
            .where( f -> f.matchAll() )
            .fetchHits( 20 ); ⑧
}
```

① Retrieve a reference to an implementation-specific datastore.
② Retrieve the `SearchMapping`.
③ Open a connection to the datastore (this is just an imaginary API, for the purpose of this example). Note we're using a try-with-resources block, so that the connection will automatically be closed when we're done with it.
④ Start creating a new session. Note we're using a try-with-resources block, so that the session will automatically be closed when we're done with it.
⑤ Pass the connection to the new session.
⑥ Build the new session.
⑦ Create a search query: since we don't use `select()`, hits will have their default representations: entities loaded from the datastore.
⑧ Retrieve the search hits as entities loaded from the datastore.

10.2.6. Programmatic mapping

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

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

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

`@SearchEntity` and its corresponding programmatic mapping `.searchEntity()` are unnecessary for Hibernate ORM entities, and in fact unsupported when using the Hibernate ORM integration.

See HSEARCH-5076 to track progress on allowing the use of `@SearchEntity` in the Hibernate ORM integration to map non-ORM entities.

You can mark a type as an entity type through the programmatic mapping too. Behavior and options are identical to annotation-based mapping.

Example 10.11: Marking a type as an entity type with `.searchEntity()`

```
TypeMappingStep bookMapping = mapping.type( Book.class );
bookMapping.searchEntity();
TypeMappingStep authorMapping = mapping.type( Author.class );
authorMapping.searchEntity().name( "MyAuthorName" );
```

10.3. Entity/index mapping

10.3.1. Basics

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

Example 10.12: Marking a class for indexing with `@Indexed`

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

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

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

By default:

- The index name will be equal to the entity name, which in Hibernate ORM is set using the `@Entity` annotation and defaults to the simple class name.
- With the Hibernate ORM integration, the identifier of indexed documents will be generated from the entity identifier. Most types commonly used for entity identifiers are supported out of the box, but for more exotic types you may need specific configuration.

  With the Standalone POJO Mapper, the identifier of indexed documents needs to be mapped explicitly.

  See Mapping the document identifier for details.
- The index won't have any field. Fields must be mapped to properties explicitly. See Mapping a property to an index field with `@GenericField`, `@FullTextField`, … for details.

10.3.2. Explicit index/backend

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

Example 10.13: Explicit index name with `@Indexed.index`

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

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

Example 10.14: Explicit backend with `@Indexed.backend`

```
@Entity
@Table(name = "\"user\"")
@Indexed(backend = "backend2")
public class User {
```

Entities indexed in different backends cannot be targeted by the same query. For example, with the mappings defined above, the following code will throw an exception because `Author` and `User` are indexed in different backends:

```
// This will fail because Author and User are indexed in different backends
searchSession.search( Arrays.asList( Author.class, User.class ) )
        .where( f -> f.matchAll() )
        .fetchHits( 20 );
```

10.3.3. Conditional indexing and routing

The mapping of an entity to an index is not always as straightforward as "this entity type goes to this index". For many reasons, but mainly for performance reasons, you may want to customize when and where a given entity is indexed:

- You may not want to index all entities of a given type: for example, prevent indexing of entities when their `status` property is set to `DRAFT` or `ARCHIVED`, because users are not supposed to search for those entities.
- You may want to route entities to a specific shard of the index: for example, route entities based on their `language` property, because each user has a specific language and only searches for entities in their language.

These behaviors can be implemented in Hibernate Search by assigning a routing bridge to the indexed entity type through `@Indexed(routingBinder = …)`.

For more information about routing bridges, see Routing bridge.

10.3.4. Programmatic mapping

You can mark an entity as indexed through the programmatic mapping too. Behavior and options are identical to annotation-based mapping.

Example 10.15: Marking a class for indexing with `.indexed()`

```
TypeMappingStep bookMapping = mapping.type( Book.class );
bookMapping.indexed();
TypeMappingStep authorMapping = mapping.type( Author.class );
authorMapping.indexed().index( "AuthorIndex" );
TypeMappingStep userMapping = mapping.type( User.class );
userMapping.indexed().backend( "backend2" );
```

10.4. Mapping the document identifier

10.4.1. Basics

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

When indexing Hibernate ORM entities, the entity identifier is used as a document identifier by default. Provided the entity identifier has a supported type, identifier mapping will work out of the box and no explicit mapping is necessary.

When using the Standalone POJO Mapper, document identifiers need to be mapped explicitly.

10.4.2. Explicit identifier mapping

Explicit identifier mapping is required in the following cases:

- Hibernate Search doesn't know about the entity identifier (e.g. when using the Standalone POJO Mapper).
- OR the document identifier is not the entity identifier.
- OR the entity identifier has a type that is not supported by default. This is the case of composite identifiers (Hibernate ORM's `@EmbeddedId`, `@IdClass`), in particular.

To select a property to map to the document identifier, just apply the `@DocumentId` annotation to that property:

Example 10.16: Mapping a property to the document identifier explicitly with `@DocumentId`

```
@Entity
@Indexed
public class Book {

    @Id
    @GeneratedValue
    private Integer id;

    @NaturalId
    @DocumentId
    private String isbn;
```

```
        public Book() {
        }

        // Getters and setters
        // ...

    }
```

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

Example 10.17: Mapping a property with unsupported type to the document identifier with `@DocumentId`

```
    @Entity
    @Indexed
    public class Book {

        @Id
        @Type(ISBNUserType.class)
        @DocumentId(identifierBridge = @IdentifierBridgeRef(type = ISBNIdentifierBridge.class))
        private ISBN isbn;

        public Book() {
        }

        // Getters and setters
        // ...

    }
```

10.4.3. Supported identifier property types

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

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

Table 10.1: Property types with built-in identifier bridges

Property type	Value of document identifiers	Limitations
`All enum types`	`name()` as a `java.lang.String`	-
`java.lang.String`	Unchanged	-
`java.lang.Character, char`	A single-character `java.lang.String`	-
`java.lang.Byte, byte`	`toString()`	-
`java.lang.Short, short`	`toString()`	-
`java.lang.Integer, int`	`toString()`	-
`java.lang.Long, long`	`toString()`	-
`java.lang.Double, double`	`toString()`	-
`java.lang.Float, float`	`toString()`	-
`java.lang.Boolean, boolean`	`toString()`	-
`java.math.BigDecimal`	`toString()`	-
`java.math.BigInteger`	`toString()`	-

Property type	Value of document identifiers	Limitations
`java.net.URI`	`toString()`	-
`java.net.URL`	`toExternalForm()`	-
`java.time.Instant`	Formatted according to `DateTimeFormatter.ISO_INSTANT`.	-
`java.time.LocalDate`	Formatted according to `DateTimeFormatter.ISO_LOCAL_DATE`.	-
`java.time.LocalTime`	Formatted according to `DateTimeFormatter.ISO_LOCAL_TIME`.	-
`java.time.LocalDateTime`	Formatted according to `DateTimeFormatter.ISO_LOCAL_DATE_TIME`.	-
`java.time.OffsetDateTime`	Formatted according to `DateTimeFormatter.ISO_OFFSET_DATE_TIME`.	-
`java.time.OffsetTime`	Formatted according to `DateTimeFormatter.ISO_OFFSET_TIME`.	-
`java.time.ZonedDateTime`	Formatted according to `DateTimeFormatter.ISO_ZONED_DATE_TIME`.	-
`java.time.ZoneId`	`getId()`	-
`java.time.ZoneOffset`	`getId()`	-
`java.time.Period`	Formatted according to the ISO 8601 format for a duration (e.g. `P1900Y12M21D`).	-
`java.time.Duration`	Formatted according to the ISO 8601 format for a duration, using seconds and nanoseconds only (e.g. `PT1.000000123S`).	-
`java.time.Year`	Formatted according to the ISO 8601 format for a Year (e.g. `2017` for 2017 AD, `0000` for 1 BC, `-10000` for 10,001 BC, etc.).	-
`java.time.YearMonth`	Formatted according to the ISO 8601 format for a Year-Month (e.g. `2017-11` for November 2017).	-
`java.time.MonthDay`	Formatted according to the ISO 8601 format for a Month-Day (e.g. `--11-06` for November 6th).	-
`java.util.UUID`	`toString()` as a `java.lang.String`	-
`java.util.Calendar`	A `java.time.ZonedDateTime` representing the same date/time and timezone, formatted according to `DateTimeFormatter.ISO_ZONED_DATE_TIME`.	See Support for legacy `java.util` date/time APIs.
`java.util.Date`	`Instant.ofEpochMilli(long)` as a `java.time.Instant` formatted according to `DateTimeFormatter.ISO_INSTANT`.	See Support for legacy `java.util` date/time APIs.
`java.sql.Timestamp`	`Instant.ofEpochMilli(long)` as a `java.time.Instant` formatted according to `DateTimeFormatter.ISO_INSTANT`.	See Support for legacy `java.util` date/time APIs.
`java.sql.Date`	`Instant.ofEpochMilli(long)` as a `java.time.Instant` formatted according to `DateTimeFormatter.ISO_INSTANT`.	See Support for legacy `java.util` date/time APIs.

Property type	Value of document identifiers	Limitations
`java.sql.Time`	`Instant.ofEpochMilli(long)` as a `java.time.Instant`, formatted according to `DateTimeFormatter.ISO_INSTANT`.	See Support for legacy `java.util` date/time APIs.
`GeoPoint` and subtypes	Latitude as double and longitude as double, separated by a comma (e.g. `41.8919, 12.51133`).	-

10.4.4. Programmatic mapping

You can map the document identifier through the programmatic mapping too. Behavior and options are identical to annotation-based mapping.

Example 10.18: Mapping a property to the document identifier explicitly with `.documentId()`

```
TypeMappingStep bookMapping = mapping.type( Book.class );
bookMapping.indexed();
bookMapping.property( "isbn" ).documentId();
```

10.5. Mapping a property to an index field with `@GenericField`, `@FullTextField`, ...

10.5.1. Basics

Properties of an entity can be mapped to an index field directly: you just need to add an annotation, configure the field through the annotation attributes, and Hibernate Search will take care of extracting the property value and populating the index field when necessary.

Mapping a property to an index field looks like this:

Example 10.19: Mapping properties to fields directly

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

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

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

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

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

③ Map another property to its own field.

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

The `@*Field` annotation

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

The type of the property

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

10.5.2. Available field annotations

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

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

`@GenericField`

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

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

`@FullTextField`

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

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

Full-text fields also allow highlighting.

Full-text fields should be assigned an analyzer, referenced by its name. By default, the analyzer named `default` will be used. See Analysis for more details about analyzers and full-text analysis. For instructions on how to change the default analyzer, see the dedicated section in the documentation of your backend: Lucene or Elasticsearch

Note you can also define a search analyzer to analyze searched terms differently.

Full-text fields cannot be sorted on nor aggregated. If you need to sort on, or aggregate on, the value of a property, it is recommended to use `@KeywordField`, with a normalizer if necessary (see below). Note that multiple fields can be added to the same property, so you can use both `@FullTextField` and `@KeywordField` if you need both full-text search and sorting: you will just need to use a distinct name for each of those two fields.

`@KeywordField`

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

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

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

`@ScaledNumberField`

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

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

This annotation allows to set the decimalScale attribute.

`@NonStandardField`

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

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

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

`@VectorField`

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

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

You are encouraged to use incubating features so the development team can get feedback and improve them,

but you should be prepared to update code which relies on them as needed.

Specific field type for vector fields to be used in a vector search.

Vector fields accept values of type `float[]` or `byte[]` and **require** that the dimension of stored vectors is specified upfront and that the indexed vectors size match this dimension.

Besides that, vector fields allow optionally configuring the similarity function used during search, `efConstruction` and `m` used during indexing.

Vector fields, on the contrary to the other field types, disable the container extraction by default Manually setting the extraction to `DEFAULT` will result in an exception. Only explicitly configured extractors are allowed for vector fields.

It is not allowed to index multiple vectors within the same field, i.e. vector fields cannot be multivalued.

10.5.3. Field annotation attributes

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

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

`name`

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

Value: `String`. The name must not contain the dot character (`.`). Defaults to the name of the property.

`sortable`

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

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

This option is not available for `@FullTextField`. See here for an explanation and some solutions.

`projectable`

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

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

The defaults are different for the Lucene and Elasticsearch backends: with Lucene, the default is `Projectable.NO`, while with Elasticsearch it's `Projectable.YES`.

For Elasticsearch if any of `projectable` or `sortable` properties are resolved to `YES` on a `GeoPoint` field then this field automatically becomes both `projectable` and `sortable` even if one of them was explicitly set to `NO`.

`aggregable`

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

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

This option is not available for `@FullTextField`. See here for an explanation and some solutions.

`searchable`

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

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

`indexNullAs`

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

Disabled by default.

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

`extraction`

How elements to index should be extracted from the property in the case of container types (`List`, `Optional`, `Map`, …).

By default, for properties that have a container type, the innermost elements will be indexed. For example for a property of type `List<String>`, elements of type `String` will be indexed.

Vector fields disable the extraction by default.

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

`analyzer`

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

By default, the analyzer named `default` will be used.

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

`searchAnalyzer`

An optional different analyzer, overriding the one defined with the `analyzer` attribute, to use only when analyzing searched terms.

If not defined, the analyzer assigned to `analyzer` will be used.

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

`normalizer`

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

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

`norms`

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

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

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

`termVector`

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

The different values of this attribute are:

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

Value	Definition
`TermVector.WITH_POSITIONS_OFFSETS_PAYLOADS`	Store the term vector, token position, offset information and token payloads. This is the same as `TermVector.WITH_POSITION_OFFSETS` plus it contains the payload of each occurrence of a term in a document.

Note that highlighter types requested by the full-text field might affect the finally resolved term vector storing strategy. Since the fast vector highlighter type has specific requirements regarding the term vector storing strategy, if it is requested explicitly or implicitly through the usage of `Highlightable.ANY`, it will set the strategy to `TermVector.WITH_POSITIONS_OFFSETS` unless a strategy was already specified. An exception will be thrown if a non-default strategy that is not compatible with the fast vector highlighter is used.

`decimalScale`

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

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

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

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

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

Note that rounding does not affect projections, which will return the original value without any loss of precision.

A typical use case is monetary amounts, with a decimal scale of 2 because only two digits are generally needed beyond the decimal point.

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

`highlightable`

Whether the field can be highlighted and if so which highlighter types can be applied to it. I.e. whether the field value is indexed/stored in a specific format to allow highlighting later when querying. Only available on `@FullTextField`.

While for most cases picking one highlighter type should be enough, this attribute can accept multiple, non contradicting values. Please refer to highlighter types section to see which highlighter to select. Available values are:

Value	Definition
`Highlightable.NO`	Do not allow highlighting on the field.
`Highlightable.ANY`	Allow any highlighter type be applied for highlighting the field.
`Highlightable.PLAIN`	Allow the plain highlighter type be applied for highlighting the field.
`Highlightable.UNIFIED`	Allow the unified highlighter type be applied for highlighting the field.
`Highlightable.FAST_VECTOR`	Allow the fast vector highlighter type be applied for highlighting the field. This highlighter type requires a term vector storage strategy to be set to `WITH_POSITIONS_OFFSETS` or `WITH_POSITIONS_OFFSETS_PAYLOADS`.

Value	Definition
`Highlightable.DEFAULT`	Use the backend-specific default that is dependent on an overall field configuration. Elasticsearch's default value is `[Highlightable.PLAIN, Highlightable.UNIFIED]`. Lucene's default value is dependent on the projectable value configured for the field. If the field is projectable then `[PLAIN, UNIFIED]` highlighters are supported. Otherwise, highlighting is not supported (`Highlightable.NO`). Additionally, if the term vector storing strategy is set to `WITH_POSITIONS_OFFSETS` or `WITH_POSITIONS_OFFSETS_PAYLOADS`, both backends would support the `FAST_VECTOR` highlighter, if they already support the other two (`[PLAIN, UNIFIED]`).

`dimension`

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

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

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

The size of the stored vectors. This is a required field. This size should match the vector size of the vectors produced by the model used to convert the data into vector representation. It is expected to be a positive integer value. Maximum accepted value is backend-specific. For the Lucene backend the dimension must be in `[1, 16000]` range. As for the Elasticsearch backend the range depends on the distribution. See the Elasticsearch/OpenSearch specific documentation to learn about the vector types of these distributions.

Only available on `@VectorField`.

`vectorSimilarity`

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

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

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

Defines how vector similarity is calculated during a vector search.

Only available on `@VectorField`.

Value	Definition
`VectorSimilarity.L2`	An L2 (Euclidean) norm, that is a sensible default for most scenarios. Distance between vectors `x` and `y` is calculated as `d(x,y) = sqrt(sum[i=1; i<n+1]( (x(i) - y(i))*(x(i) - y(i)) )` and the score function is `s = 1/(1+d*d)`
`VectorSimilarity.DOT_PRODUCT`	Inner product (dot product in particular). Distance between vectors `x` and `y` is calculated as `d(x,y) = sum[i=1; i< n+1] ( x(i)*y(i) ) ` and the score function is `s = 1/(1+d)` To use this similarity efficiently, both index and search vectors **must** be normalized; otherwise search may produce poor results. Floating point vectors must be normalized to be of unit length, while byte vectors should simply all have the same norm.

Value	Definition
`VectorSimilarity.COSINE`	Cosine similarity. Distance between vectors `x` and `y` is calculated as `d(x,y) = (1 - sum[i=1; i<n+1] ( x(i)*y(i) )/( sqrt( sum[i=1; i<n+1] x(i)*x(i) ) sqrt( sum[i=1; i<n+1] y(i)*y(i) ) ) )` and the score function is `s = 1/(1+d)`
`VectorSimilarity.MAX_INNER_PRODUCT`	Similar to a dot product similarity, but does not require vector normalization. Distance between vectors `x` and `y` is calculated as `d(x,y) = sum[i=1; i<n+1] ( x(i)*y(i) )` and the score function is `d<0 ? 1/(1-d) : d+1`
`VectorSimilarity.DEFAULT`	Use the backend-specific default. For the Lucene backend an `L2` similarity is used.

Table 4. How the vector similarity matches to a backend-specific value

Hibernate Search Value	Lucene Backend	Elasticsearch Backend	Elasticsearch Backend (OpenSearch distribution)
`DEFAULT`	`EUCLIDEAN`	Elasticsearch default	OpenSearch default.
`L2`	`EUCLIDEAN`	`l2_norm`	`l2`
`DOT_PRODUCT`	`DOT_PRODUCT`	`dot_product`	currently **not supported** by OpenSearch and will result in an exception.
`COSINE`	`COSINE`	`cosine`	`cosinesimil`
`MAX_INNER_PRODUCT`	`MAXIMUM_INNER_PRODUCT`	`max_inner_product`	currently **not supported** by OpenSearch and will result in an exception.

`efConstruction`

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

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

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

`efConstruction` is the size of the dynamic list used during k-NN graph creation. It affects how vectors are stored. Higher values lead to a more accurate graph but slower indexing speed.

Default value is backend-specific.

Only available on `@VectorField`.

`m`

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

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

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

The number of neighbors each node will be connected to in the HNSW (Hierarchical Navigable Small World graphs) graph. Modifying this value will have an impact on memory consumption. It is recommended to keep this value between 2 and 100.

Default value is backend-specific.

Only available on `@VectorField`.

10.5.4. Supported property types

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

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

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

Table 10.2: Property types with built-in value bridges

Property type	Value of index field (if different)	Limitations	Parsing method for 'indexNullAs'/terms in query string predicates
`All enum types`	`name()` as a `java.lang.String`	-	`Enum.valueOf(String)`
`java.lang.String`	-	-	-
`java.lang.Character, char`	A single-character `java.lang.String`	-	Accepts any single-character `java.lang.String`
`java.lang.Byte, byte`	-	-	`Byte.parseByte(String)`
`java.lang.Short, short`	-	-	`Short.parseShort(String)`
`java.lang.Integer, int`	-	-	`Integer.parseInt(String)`
`java.lang.Long, long`	-	-	`Long.parseLong(String)`
`java.lang.Double, double`	-	-	`Double.parseDouble(String)`
`java.lang.Float, float`	-	-	`Float.parseFloat(String)`
`java.lang.Boolean, boolean`	-	-	Accepts the strings `true` or `false`, ignoring case
`java.math.BigDecimal`	-	-	`new BigDecimal(String)`
`java.math.BigInteger`	-	-	`new BigInteger(String)`
`java.net.URI`	`toString()` as a `java.lang.String`	-	`new URI(String)`
`java.net.URL`	`toExternalForm()` as a `java.lang.String`	-	`new URL(String)`
`java.time.Instant`	-	Possibly lower range/resolution	`Instant.parse(String)`
`java.time.LocalDate`	-	Possibly lower range/resolution	`LocalDate.parse(String).`
`java.time.LocalTime`	-	Possibly lower range/resolution	`LocalTime.parse(String)`
`java.time.LocalDateTime`	-	Possibly lower range/resolution	`LocalDateTime.parse(String)`
`java.time.OffsetDateTime`	-	Possibly lower range/resolution	`OffsetDateTime.parse(String)`
`java.time.OffsetTime`	-	Possibly lower range/resolution	`OffsetTime.parse(String)`
`java.time.ZonedDateTime`	-	Possibly lower range/resolution	`ZonedDateTime.parse(String)`
`java.time.ZoneId`	`getId()` as a `java.lang.String`	-	`ZoneId.of(String)`

Property type	Value of index field (if different)	Limitations	Parsing method for 'indexNullAs'/terms in query string predicates
`java.time.ZoneOffset`	`getTotalSeconds()` as a `java.lang.Integer`	-	`ZoneOffset.of(String)`
`java.time.Period`	A formatted `java.lang.String`: `<years on 11 characters><months on 11 characters><days on 11 characters>`	-	`Period.parse(String)`
`java.time.Duration`	`toNanos()` as a `java.lang.Long`	Possibly lower range/resolution	`Duration.parse(String)`
`java.time.Year`	-	Possibly lower range/resolution	`Year.parse(String)`
`java.time.YearMonth`	-	Possibly lower range/resolution	`YearMonth.parse(String)`
`java.time.MonthDay`	-	-	`MonthDay.parse(String)`
`java.util.UUID`	`toString()` as a `java.lang.String`	-	`UUID.fromString(String)`
`java.util.Calendar`	A `java.time.ZonedDateTime` representing the same date/time and timezone.	See Support for legacy `java.util` date/time APIs.	`ZonedDateTime.parse(String)`
`java.util.Date`	`Instant.ofEpochMilli(long)` as a `java.time.Instant`.	See Support for legacy `java.util` date/time APIs.	`Instant.parse(String)`
`java.sql.Timestamp`	`Instant.ofEpochMilli(long)` as a `java.time.Instant`.	See Support for legacy `java.util` date/time APIs.	`Instant.parse(String)`
`java.sql.Date`	`Instant.ofEpochMilli(long)` as a `java.time.Instant`.	See Support for legacy `java.util` date/time APIs.	`Instant.parse(String)`
`java.sql.Time`	`Instant.ofEpochMilli(long)` as a `java.time.Instant`.	See Support for legacy `java.util` date/time APIs.	`Instant.parse(String)`
`GeoPoint` and subtypes	-	-	Latitude as double and longitude as double, separated by a comma. Ex: `41.8919, 12.51133`.

Range and resolution of date/time fields

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

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

The documentation of each backend provides more information: see here for Lucene and here for Elasticsearch.

10.5.5. Support for legacy `java.util` date/time APIs

Using legacy date/time types such as `java.util.Calendar`, `java.util.Date`, `java.sql.Timestamp`, `java.sql.Date`, `java.sql.Time` is not recommended, due to their numerous quirks and shortcomings. The `java.time` package introduced in Java 8 should generally be preferred.

That being said, integration constraints may force you to rely on the legacy date/time APIs, which is why Hibernate Search still attempts to support them on a best effort basis.

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

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

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

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

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

> Technically, conversions are difficult because the `java.time` APIs and the legacy date/time APIs do not have the same internal calendar.
>
> In particular:
>
> - `java.time` assumes a "Local Mean Time" before 1900, while legacy date/time APIs do not support it (JDK-6281408), As a result, time values (number of milliseconds since the epoch) reported by the two APIs will be different for dates before 1900.
> - `java.time` uses a proleptic Gregorian calendar before October 15, 1582, meaning it acts as if the Gregorian calendar, along with its system of leap years, had always existed. Legacy date/time APIs, on the other hand, use the Julian calendar before that date (by default), meaning the leap years are not exactly the same ones. As a result, some dates that are deemed valid by one API will be deemed invalid by the other, for example February 29, 1500.
>
> Those are the two main problems, but there may be others.

10.5.6. Mapping custom property types

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

There are two cases to distinguish between:

1. If the unsupported type is simply a container (`List<String>`) or multiple nested containers (`Map<Integer, List<String>>`) whose elements have a supported type, then what you need is a container extractor. See Mapping container types with container extractors for more information.
2. Otherwise, you will have to rely on a custom component, called a bridge, to extract data from your type. See Binding and bridges for more information on custom bridges.

10.5.7. Programmatic mapping

You can map properties of an entity to an index field directly through the programmatic mapping too. Behavior and options are identical to annotation-based mapping.

Example 10.20: Mapping properties to fields directly with `.genericField()`, `.fullTextField()`, …

```
TypeMappingStep bookMapping = mapping.type( Book.class );
bookMapping.indexed();
bookMapping.property( "title" )
        .fullTextField()
                .analyzer( "english" ).projectable( Projectable.YES )
        .keywordField( "title_sort" )
                .normalizer( "english" ).sortable( Sortable.YES );
bookMapping.property( "pageCount" )
        .genericField().projectable( Projectable.YES ).sortable( Sortable.YES );
```

10.6. Mapping associated elements with `@IndexedEmbedded`

10.6.1. Basics

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

Example 10.21: A multi-entity model with associations

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

- `title`
- … and nothing else.

```
@Entity
@Indexed ①
public class Book {

    @Id
    private Integer id;

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

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

    public Book() {
    }

    // Getters and setters
    // ...

}
```

```
@Entity
public class Author {

    @Id
    private Integer id;

    private String name;

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

    public Author() {
    }

    // Getters and setters
    // ...

}
```

① The `Book` entity is indexed.
② The `title` of the book is mapped to an index field.
③ But how to index the `Author` name into the `Book` index?

When searching for a book, users will likely need to search by author name. In the world of high-performance indexes, cross-index joins are costly and usually not an option. The best way to address such use cases is generally to copy data: when indexing a `Book`, just copy the name of all its authors into the `Book` document.

That's what `@IndexedEmbedded` does: it instructs Hibernate Search to *embed* the fields of an associated object into the main object. In the example below, it will instruct Hibernate Search to embed the `name` field defined in `Author` into `Book`, creating the field `authors.name`.

`@IndexedEmbedded` can be used on Hibernate ORM's `@Embedded` properties as well as associations (`@OneToOne`, `@OneToMany`, `@ManyToMany`, …).

Example 10.22: Using `@IndexedEmbedded` to index associated elements

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

- `title`
- `authors.name`

```
@Entity
@Indexed
public class Book {

    @Id
    private Integer id;

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

    @ManyToMany
    @IndexedEmbedded ①
    private List<Author> authors = new ArrayList<>();

    public Book() {
    }

    // Getters and setters
    // ...

}
```

```
@Entity
public class Author {

    @Id
    private Integer id;

    @FullTextField(analyzer = "name") ②
    private String name;

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

    public Author() {
    }

    // Getters and setters
    // ...

}
```

① Add an `@IndexedEmbedded` to the `authors` property.

② Map `Author.name` to an index field, even though `Author` is not directly mapped to an index (no `@Indexed`).

Document identifiers are not index fields. Consequently, they will be ignored by `@IndexedEmbedded`.

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

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

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

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

Example 10.23: Nesting multiple `@IndexedEmbedded`

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

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

```
@Entity
@Indexed
public class Book {

    @Id
    private Integer id;

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

    @ManyToMany
    @IndexedEmbedded ①
    private List<Author> authors = new ArrayList<>();

    public Book() {
    }

    // Getters and setters
    // ...

}
```

```
@Entity
public class Author {

    @Id
    private Integer id;

    @FullTextField(analyzer = "name") ②
    private String name;

    @Embedded
    @IndexedEmbedded ③
    private Address placeOfBirth;

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

    public Author() {
    }

    // Getters and setters
    // ...

}
```

```
@Embeddable
public class Address {

    @FullTextField(analyzer = "name") ④
    private String country;

    private String city;

    private String street;

    public Address() {
    }
```

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

    }
```

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

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

To address this, see Filtering embedded fields and breaking `@IndexedEmbedded` cycles.

10.6.2. `@IndexedEmbedded` and `null` values

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

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

10.6.3. `@IndexedEmbedded` on container types

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

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

10.6.4. Setting the object field name with `name`

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

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

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

10.6.5. Setting the field name prefix with `prefix`

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

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

It is possible to change this prefix by setting the `prefix` attribute, for example `@IndexedEmbedded(prefix = "author.")` (do not forget the trailing dot!).

The prefix should generally be a sequence of non-dots ending with a single dot, for example `my_Property.`.

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

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

10.6.6. Casting the target of `@IndexedEmbedded` with `targetType`

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

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

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

10.6.7. Reindexing when embedded elements change

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

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

When Hibernate Search is unable to handle an association, it will throw an exception on bootstrap. If this happens, refer to Basics to know more.

10.6.8. Embedding the entity identifier

Mapping a property as an identifier in the indexed-embedded type will not automatically result into it being embedded when using `@IndexedEmbedded` on that type, because document identifiers are not fields.

To embed the data of such a property, you can use `@IndexedEmbedded(includeEmbeddedObjectId = true)`, which will have Hibernate Search automatically insert a field in the resulting embedded object for the indexed-embedded type's identifier property.

The index field will be defined as if the following field annotation was put on the identifier property of the embedded type: `@GenericField(searchable = Searchable.YES, projectable = Projectable.YES)`. The name of the index field will be the name of the identifier property. Its bridge will be the identifier bridge referenced by the embedded type's `@DocumentId` annotation, if any, or the default value bridge for the identifier property type's, by default.

If you need more advanced mapping (custom name, custom bridge, sortable, …), do not use `includeEmbeddedObjectId`.

Instead, define the field explicitly in the indexed-embedded type by annotating the identifier property with `@GenericField` or a similar field annotation, and make sure the field is included by `@IndexedEmbedded` by configuring filters as necessary.

Below is an example of using `includeEmbeddedObjectId`:

Example 10.24: Including indexed-embedded IDs with `includeEmbeddedObjectId`

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

- `name`
- `department.name`: implicitly included by `@IndexedEmbedded`.
- `department.id`: explicitly inserted by `includeEmbeddedObjectId = true.

```
@Entity
public class Department {

    @Id
    private Integer id; ①

    @FullTextField
    private String name;

    @OneToMany(mappedBy = "department")
    private List<Employee> employees = new ArrayList<>();

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

```
}
```

① The `Department` identifier is not mapped to an index field (not `@*Field` annotation).

```
@Entity
@Indexed
public class Employee {

    @Id
    private Integer id;

    @FullTextField
    private String name;

    @ManyToOne
    @IndexedEmbedded(includeEmbeddedObjectId = true) ①
    private Department department;

    // Getters and setters
    // ...

}
```

① `@IndexedEmbedded` will insert a `department.id` field into the `Employee` index for the `Department` identifier, even though in `Department` the identifier property is not mapped to an index field.

10.6.9. Filtering embedded fields and breaking `@IndexedEmbedded` cycles

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

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

- If the indexed-embedded element declares many index fields (Hibernate Search fields), only some of which are actually useful to the "index-embedding" type, the extra fields will decrease indexing performance needlessly.
- If there is a cycle of `@IndexedEmbedded` (e.g. `A` index-embeds `b` of type `B`, which index-embeds `a` of type `A`) the index-embedding type will end up with an infinite amount of fields (`a.b.someField`, `a.b.a.b.someField`, `a.b.a.b.a.b.someField`, …), which Hibernate Search will detect and reject with an exception.

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

`includePaths`

: The paths of index fields from the indexed-embedded element that should be embedded.

  Provided paths must be relative to the indexed-embedded element, i.e. they must not include its name or prefix.

  This takes precedence over `includeDepth` (see below).

  Cannot be used in combination with `excludePaths` in the same `@IndexedEmbedded`.

`excludePaths`

: The paths of index fields from the indexed-embedded element that must **not** be embedded.

  Provided paths must be relative to the indexed-embedded element, i.e. they must not include its name or prefix.

  This takes precedence over `includeDepth` (see below).

  Cannot be used in combination with `includePaths` in the same `@IndexedEmbedded`.

`includeDepth`

: The number of levels of indexed-embedded that will have all their fields included by default.

  `includeDepth` is the number of `@IndexedEmbedded` that will be traversed and for which all fields of the indexed-embedded element will be included, even if these fields are not included explicitly through `includePaths`, unless these fields are excluded explicitly through `excludePaths`:

- `includeDepth=0` means that fields of this indexed-embedded element are **not** included, nor is any field of nested indexed-embedded elements, unless these fields are included explicitly through `includePaths`.
- `includeDepth=1` means that fields of this indexed-embedded element **are** included, unless these fields are excluded explicitly through `excludePaths`, but **not** fields of nested indexed-embedded elements (`@IndexedEmbedded` within this `@IndexedEmbedded`), unless these fields are included explicitly through `includePaths`.
- `includeDepth=2` means that fields of this indexed-embedded element and fields of the immediately nested indexed-embedded (`@IndexedEmbedded` within this `@IndexedEmbedded`) elements **are** included, unless these fields are explicitly excluded through `excludePaths`, but **not** fields of nested indexed-embedded elements beyond that (`@IndexedEmbedded` within an `@IndexedEmbedded` within this `@IndexedEmbedded`), unless these fields are included explicitly through `includePaths`.
- And so on.

The default value depends on the value of the `includePaths` attribute:

- if `includePaths` is empty, the default is `Integer.MAX_VALUE` (include all fields at every level)
- if `includePaths` is **not** empty, the default is `0` (only include fields included explicitly).

Dynamic fields and filtering

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

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

Mixing `includePaths` and `excludePaths` at different nesting levels

In general, it is possible to use `includePaths` and `excludePaths` at different levels of nested `@IndexedEmbedded`. When doing so, keep in mind that the filter at each level can only reference reachable paths, i.e. a filter cannot reference a path that was excluded by a nested `@IndexedEmbedded` (implicitly or explicitly).

Below are three examples: one leveraging `includePaths` only, one leveraging `excludePaths`, and one leveraging `includePaths` and `includeDepth`.

Example 10.25: Filtering indexed-embedded fields with `includePaths`

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

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

The following fields in particular are excluded:

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

```
@Entity
@Indexed
public class Human {

    @Id
    private Integer id;

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

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

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

    @ManyToMany(mappedBy = "parents")
```

```
        private List<Human> children = new ArrayList<>();

        public Human() {
        }

        // Getters and setters
        // ...

    }
```

Example 10.26: Filtering indexed-embedded fields with `excludePaths`

This mapping will result in the same schema as in the Filtering indexed-embedded fields with `includePaths` example, but through using the `excludePaths` instead. Following fields in the `Human` index will be declared:

- `name`
- `nickname`
- `parents.name`: implicitly included because `includeDepth` on `parents` defaults to `Integer.MAX_VALUE`.
- `parents.nickname`: implicitly included because `includeDepth` on `parents` defaults to `Integer.MAX_VALUE`.
- `parents.parents.name`: implicitly included because `includeDepth` on `parents` defaults to `Integer.MAX_VALUE`.

The following fields in particular are excluded:

- `parents.parents.nickname`: **not** included because `excludePaths` explicitly excludes `parents.nickname`.
- `parents.parents.parents`/`parents.parents.parents.<any-field>`: **not** included because `excludePaths` explicitly excludes `parents.parents` stopping any further traversing.

```
    @Entity
    @Indexed
    public class Human {

        @Id
        private Integer id;

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

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

        @ManyToMany
        @IndexedEmbedded(excludePaths = { "parents.nickname", "parents.parents" })
        private List<Human> parents = new ArrayList<>();

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

        public Human() {
        }

        // Getters and setters
        // ...

    }
```

Example 10.27: Filtering indexed-embedded fields with `includePaths` *and* `includeDepth`

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

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

- `parents.parents.name`: implicitly included at depth `1` because `includeDepth > 1` (so `parents.parents.*` is included implicitly).
- `parents.parents.nickname`: implicitly included at depth `1` because `includeDepth > 1` (so `parents.parents.*` is included implicitly).
- `parents.parents.parents.name`: **not** implicitly included at depth `2` because `includeDepth = 2` (so `parents.parents.parents` is included implicitly, but subfields can only be included explicitly) but explicitly included because `includePaths` on `parents` includes `parents.parents.name`.

The following fields in particular are excluded:

- `parents.parents.parents.nickname`: **not** implicitly included at depth `2` because `includeDepth = 2` (so `parents.parents.parents` is included implicitly, but subfields must be included explicitly) and **not** explicitly included either because `includePaths` on `parents` does not include `parents.parents.nickname`.
- `parents.parents.parents.parents.name`: **not** implicitly included at depth `3` because `includeDepth = 2` (so `parents.parents.parents` is included implicitly, but `parents.parents.parents.parents` and subfields can only be included explicitly) and **not** explicitly included either because `includePaths` on `parents` does not include `parents.parents.parents.name`.

```
@Entity
@Indexed
public class Human {

    @Id
    private Integer id;

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

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

    @ManyToMany
    @IndexedEmbedded(includeDepth = 2, includePaths = { "parents.parents.name" })
    private List<Human> parents = new ArrayList<>();

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

    public Human() {
    }

    // Getters and setters
    // ...

}
```

10.6.10. Structuring embedded elements as nested documents using `structure`

Indexed-embedded fields can be structured in one of two ways, configured through the `structure` attribute of the `@IndexedEmbedded` annotation. To illustrate structure options, let's assume the class `Book` is annotated with `@Indexed` and its `authors` property is annotated with `@IndexedEmbedded`:

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

`DEFAULT` or `FLATTENED` structure

By default, or when using `@IndexedEmbedded(structure = FLATTENED)` as shown below, indexed-embedded fields are "flattened", meaning that the tree structure is not preserved.

Example 10.28: `@IndexedEmbedded` with a flattened structure

```
@Entity
@Indexed
public class Book {

    @Id
    private Integer id;

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

    @ManyToMany
    @IndexedEmbedded(structure = ObjectStructure.FLATTENED) ①
    private List<Author> authors = new ArrayList<>();

    public Book() {
    }

    // Getters and setters
    // ...

}
```

① Explicitly set the structure of indexed-embedded to `FLATTENED`. This is not strictly necessary, since `FLATTENED` is the default.

```
@Entity
public class Author {

    @Id
    private Integer id;

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

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

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

    public Author() {
    }

    // Getters and setters
    // ...

}
```

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

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

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

This is more efficient for indexing and querying, but can cause unexpected behavior when querying the index on both the author's first name and the author's last name.

For example, the book instance described above **would** show up as a match to a query such as `authors.firstname:Ty AND authors.lastname:Abraham`, even though "Ty Abraham" is not one of this book's authors:

Example 10.29: Searching with a flattened structure

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.and(
                f.match().field( "authors.firstName" ).matching( "Ty" ), ①
                f.match().field( "authors.lastName" ).matching( "Abraham" ) ①
        ) )
        .fetchHits( 20 );

assertThat( hits ).isNotEmpty(); ②
```

① Require that hits have an author with the first name `Ty` and an author with the last name `Abraham`… but not necessarily the same author!

② The hits will include a book whose authors are "Ty Daniel" and "Frank Abraham".

`NESTED` structure

When indexed-embedded elements are "nested", i.e. when using `@IndexedEmbedded(structure = NESTED)` as shown below, the tree structure is preserved by transparently creating one separate "nested" document for each indexed-embedded element.

Example 10.30: `@IndexedEmbedded` with a nested structure

```
@Entity
@Indexed
public class Book {

    @Id
    private Integer id;

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

    @ManyToMany
    @IndexedEmbedded(structure = ObjectStructure.NESTED) ①
    private List<Author> authors = new ArrayList<>();

    public Book() {
    }

    // Getters and setters
    // ...

}
```

① Explicitly set the structure of indexed-embedded objects to `NESTED`.

```
@Entity
public class Author {

    @Id
    private Integer id;

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

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

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

    public Author() {
    }

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

```
}
```

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

- Book document
  - title = Leviathan Wakes
  - Nested documents
    - Nested document #1 for "authors"
      - authors.firstName = Daniel
      - authors.lastName = Abraham
    - Nested document #2 for "authors"
      - authors.firstName = Ty
      - authors.lastName = Frank

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

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

If special care is taken when building predicates on fields within nested documents, using a `nested` predicate, queries containing predicates on both the author's first name and the author's last name will behave as one would (intuitively) expect.

For example, the book instance described above would **not** show up as a match to a query such as `authors.firstname:Ty AND authors.lastname:Abraham`, thanks to the `nested` predicate (which can only be used when indexing with the `NESTED` structure):

Example 10.31: Searching with a nested structure

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.nested( "authors" ) ①
                .add( f.match().field( "authors.firstName" ).matching( "Ty" ) ) ②
                .add( f.match().field( "authors.lastName" ).matching( "Abraham" ) ) ) ②
        .fetchHits( 20 );

assertThat( hits ).isEmpty(); ③
```

① Require that the two constraints (first name and last name) apply to the **same** author.
② Require that hits have an author with the first name `Ty` and an author with the last name `Abraham`.
③ The hits will **not** include a book whose authors are "Ty Daniel" and "Frank Abraham".

With the Lucene backend, the nested structure is also necessary if you want to perform `object` projections.

10.6.11. Filtering association elements

Sometimes, only some elements of an association should be included in an `@IndexedEmbedded`.

For example a `Book` entity might index-embed `BookEdition` instances, but some editions might be retired and thus need to be filtered out before indexing.

Such filtering can be achieved by applying `@IndexedEmbedded` to a transient getter representing the filtered association, and configuring reindexing with `@AssociationInverseSide` and `@IndexingDependency.derivedFrom`.

Example 10.32: Filtered an `@IndexedEmbedded` association with a transient getter, `@AssociationInverseSide` and `@IndexingDependency.derivedFrom`

```
@Entity
@Indexed
public class Book {

    @Id
```

```
    private Integer id;

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

    @OneToMany(mappedBy = "book")
    @OrderBy("id asc")
    private List<BookEdition> editions = new ArrayList<>(); ①

    public Book() {
    }

    // Getters and setters
    // ...


    @Transient ②
    @IndexedEmbedded ③
    @AssociationInverseSide(inversePath = @ObjectPath({ ④
            @PropertyValue(propertyName = "book")
    }))
    @IndexingDependency(derivedFrom = @ObjectPath({ ⑤
            @PropertyValue(propertyName = "editions"),
            @PropertyValue(propertyName = "status")
    }))
    public List<BookEdition> getEditionsNotRetired() {
        return editions.stream()
                .filter( e -> e.getStatus() != BookEdition.Status.RETIRED )
                .collect( Collectors.toList() );
    }
}
```

```
@Entity
public class BookEdition {

    public enum Status {
        PUBLISHING,
        RETIRED
    }

    @Id
    private Integer id;

    @ManyToOne
    private Book book;

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

    private Status status; ⑥

    public BookEdition() {
    }

    // Getters and setters
    // ...

}
```

① The association between `Book` and `BookEdition` is mapped in Hibernate ORM, but not Hibernate Search.

② The transient `editionsNotRetired` property dynamically returns the editions that are not retired.

③ `@IndexedEmbedded` is applied to `editionsNotRetired` instead of `editions`. If we wanted to, we could use `@IndexedEmbedded(name = "editions")` to make this transparent when searching.

④ Hibernate ORM does not know about `editionsNotRetired`, so Hibernate Search cannot infer the inverse side of this "filtered" association. Thus, we use `@AssociationInverseSide` to tell Hibernate Search that. Should the label of a `BookEdition` be modified, Hibernate Search will use this information to retrieve the corresponding `Book` to reindex.

⑤ We use `@IndexingDependency.derivedFrom` to tell Hibernate Search that whenever the status of an edition changes, the result of `getEditionsNotRetired()` may have changed as well, requiring reindexing.

⑥ While `BookEdition#status` is not annotated, Hibernate Search will still track its changes because of the `@IndexingDependency` annotation in `Book`.

10.6.12. Programmatic mapping

You can embed the fields of an associated object into the main object through the programmatic mapping too. Behavior and options are identical to annotation-based mapping.

Example 10.33: Using `.indexedEmbedded()` to index associated elements

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

- `title`
- `authors.name`

```
TypeMappingStep bookMapping = mapping.type( Book.class );
bookMapping.indexed();
bookMapping.property( "title" )
        .fullTextField().analyzer( "english" );
bookMapping.property( "authors" )
        .indexedEmbedded();
TypeMappingStep authorMapping = mapping.type( Author.class );
authorMapping.property( "name" )
        .fullTextField().analyzer( "name" );
```

10.7. Mapping container types with container extractors

10.7.1. Basics

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

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

Same goes for other field annotations such as `@FullTextField`, as well as `@IndexedEmbedded` in particular. With `@VectorField` being an exception to this behaviour, requiring explicit instructions to extract values from a container.

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

10.7.2. Explicit container extraction

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

All built-in extractor names are available as constants in `org.hibernate.search.mapper.pojo.extractor.builtin.BuiltinContainerExtractors`.

Example 10.34: Mapping `Map` keys to an index field using explicit container extractor definition

```
@ElementCollection ①
@JoinTable(name = "book_pricebyformat")
@MapKeyColumn(name = "format")
@Column(name = "price")
@OrderBy("format asc")
@GenericField( ②
        name = "availableFormats",
        extraction = @ContainerExtraction(BuiltinContainerExtractors.MAP_KEY) ③
```

```
    )
    private Map<BookFormat, BigDecimal> priceByFormat = new LinkedHashMap<>();
```

① This annotation — and those below — are just Hibernate ORM configuration.
② Declare an index field based on the `priceByFormat` property.
③ By default, Hibernate Search would index the map values (the book prices). This uses the `extraction` attribute to specify that map keys (the book formats) must be indexed instead.

When multiple levels of extractions are necessary, multiple extractors can be configured: `extraction = @ContainerExtraction(BuiltinContainerExtractors.MAP_KEY, BuiltinContainerExtractors.OPTIONAL)`. However, such complex mappings are unlikely since they are generally not supported by Hibernate ORM.

It is possible to implement and use custom container extractors, but at the moment Hibernate Search will not detect that the changes to the data inside such container must trigger the reindexing of a containing element. Hence, the corresponding property must disable reindexing on change.

See HSEARCH-3688 for more information.

10.7.3. Disabling container extraction

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

Example 10.35: Disabling container extraction

```
    @ManyToMany
    @GenericField( ①
            name = "authorCount",
            valueBridge = @ValueBridgeRef(type = MyCollectionSizeBridge.class), ②
            extraction = @ContainerExtraction(extract = ContainerExtract.NO) ③
    )
    private List<Author> authors = new ArrayList<>();
```

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

10.7.4. Programmatic mapping

You can pick the container extractors to use explicitly when defining fields or indexed-embeddeds through the programmatic mapping too. Behavior and options are identical to annotation-based mapping.

Example 10.36: Mapping `Map` keys to an index field using `.extractor(…)`/`.extactors(…)` for explicit container extractor definition

```
    bookMapping.property( "priceByFormat" )
            .genericField( "availableFormats" )
                    .extractor( BuiltinContainerExtractors.MAP_KEY );
```

Similarly, you can disable container extraction.

Example 10.37: Disabling container extraction with `.noExtractors()`

```
    bookMapping.property( "authors" )
            .genericField( "authorCount" )
                    .valueBridge( new MyCollectionSizeBridge() )
                    .noExtractors();
```

10.8. Mapping geo-point types

10.8.1. Basics

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

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

Two options are available:

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

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

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

Example 10.38: Mapping spatial coordinates by implementing `GeoPoint` and using `@GenericField`

```
@Embeddable
public class MyCoordinates implements GeoPoint { ①

    @Basic
    private Double latitude;

    @Basic
    private Double longitude;

    protected MyCoordinates() {
        // For Hibernate ORM
    }

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

    @Override
    public double latitude() { ②
        return latitude;
    }

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

```
@Entity
@Indexed
public class Author {

    @Id
    @GeneratedValue
    private Integer id;

    private String name;

    @Embedded
    @GenericField ③
    private MyCoordinates placeOfBirth;
```

```
    public Author() {
    }

    // Getters and setters
    // ...

}
```

① Model the geo-point as an embeddable implementing `GeoPoint`. A custom type with a corresponding Hibernate ORM `UserType` would work as well.
② The geo-point type **must be immutable**: it does not declare any setter.
③ Apply the `@GenericField` annotation to the `placeOfBirth` property holding the coordinates. An index field named `placeOfBirth` will be added to the index. Options generally used on `@GenericField` can be used here as well.

The geo-point type **must be immutable**, i.e. the latitude and longitude of a given instance may never change.

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

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

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

10.8.3. Using `@GeoPointBinding`, `@Latitude` and `@Longitude`

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

Example 10.39: Mapping spatial coordinates using `@GeoPointBinding`

```
@Entity
@Indexed
@GeoPointBinding(fieldName = "placeOfBirth") ①
public class Author {

    @Id
    @GeneratedValue
    private Integer id;

    private String name;

    @Latitude ②
    private Double placeOfBirthLatitude;

    @Longitude ③
    private Double placeOfBirthLongitude;

    public Author() {
    }

    // Getters and setters
    // ...

}
```

① Apply the `@GeoPointBinding` annotation to the type, setting `fieldName` to the name of the index field.
② Apply `@Latitude` to the property holding the latitude. It must be of `double` or `Double` type.
③ Apply `@Longitude` to the property holding the longitude. It must be of `double` or `Double` type.

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

Example 10.40: Mapping spatial coordinates using `@GeoPointBinding` on a property

```
@Embeddable
public class MyCoordinates { ①

    @Basic
    @Latitude ②
    private Double latitude;

    @Basic
    @Longitude ③
    private Double longitude;

    protected MyCoordinates() {
        // For Hibernate ORM
    }

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

    public double getLatitude() {
        return latitude;
    }

    public void setLatitude(Double latitude) { ④
        this.latitude = latitude;
    }

    public double getLongitude() {
        return longitude;
    }

    public void setLongitude(Double longitude) {
        this.longitude = longitude;
    }
}
```

```
@Entity
@Indexed
public class Author {

    @Id
    @GeneratedValue
    private Integer id;

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

    @Embedded
    @GeoPointBinding ⑤
    private MyCoordinates placeOfBirth;

    public Author() {
    }

    // Getters and setters
    // ...

}
```

① Model the geo-point as embeddable. An entity would work as well.

② In the geo-point type, apply `@Latitude` to the property holding the latitude.

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

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

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

It is possible to handle multiple sets of coordinates by applying the annotations multiple times and setting the `markerSet` attribute to a unique value:

Example 10.41: Mapping multiple sets of spatial coordinates using `@GeoPointBinding`

```
@Entity
@Indexed
@GeoPointBinding(fieldName = "placeOfBirth", markerSet = "birth") ①
@GeoPointBinding(fieldName = "placeOfDeath", markerSet = "death") ②
public class Author {

    @Id
    @GeneratedValue
    private Integer id;

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

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

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

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

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

    public Author() {
    }

    // Getters and setters
    // ...

}
```

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

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

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

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

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

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

10.8.4. Programmatic mapping

You can map geo-point fields document identifier through the programmatic mapping too. Behavior and options are identical to annotation-based mapping.

Example 10.42: Mapping spatial coordinates by implementing `GeoPoint` and using `.genericField()`

```
TypeMappingStep authorMapping = mapping.type( Author.class );
authorMapping.indexed();
```

```
authorMapping.property( "placeOfBirth" )
        .genericField();
```

Example 10.43: Mapping spatial coordinates using `GeoPointBinder`

```
TypeMappingStep authorMapping = mapping.type( Author.class );
authorMapping.indexed();
authorMapping.binder( GeoPointBinder.create().fieldName( "placeOfBirth" ) );
authorMapping.property( "placeOfBirthLatitude" )
        .marker( GeoPointBinder.latitude() );
authorMapping.property( "placeOfBirthLongitude" )
        .marker( GeoPointBinder.longitude() );
```

10.9. Mapping multiple alternatives

10.9.1. Basics

In some situations, it is necessary for a particular property to be indexed differently depending on the value of another property.

For example there may be an entity that has text properties whose content is in a different language depending on the value of another property, say `language`. In that case, you probably want to analyze the text differently depending on the language.

While this could definitely be solved with a custom type bridge, a convenient solution to that problem is to use the `AlternativeBinder`. This binder solves the problem this way:

- at bootstrap, declare one index field per language, assigning a different analyzer to each field;
- at runtime, put the content of the text property in a different field based on the language.

In order to use this binder, you will need to:

- annotate a property with `@AlternativeDiscriminator` (e.g. the `language` property);
- implement an `AlternativeBinderDelegate` that will declare the index fields (e.g. one field per language) and create an `AlternativeValueBridge`. This bridge is responsible for passing the property value to the relevant field at runtime.
- apply the `AlternativeBinder` to the type hosting the properties (e.g. the type declaring the `language` property and the multi-language text properties). Generally you will want to create your own annotation for that.

Below is an example of how to use the binder.

Example 10.44: Mapping a property to a different index field based on a `language` *property using* `AlternativeBinder`

```
public enum Language { ①

    ENGLISH( "en" ),
    FRENCH( "fr" ),
    GERMAN( "de" );

    public final String code;

    Language(String code) {
        this.code = code;
    }
}
```

① A `Language` enum defines supported languages.

```
@Entity
@Indexed
public class BlogEntry {

    @Id
    private Integer id;

    @AlternativeDiscriminator ①
    @Enumerated(EnumType.STRING)
```

```
    private Language language;

    @MultiLanguageField ②
    private String text;

    // Getters and setters
    // ...

}
```

① Mark the `language` property as the discriminator which will be used to determine the language.

② Map the `text` property to multiple fields using a custom annotation.

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

    String name() default ""; ⑤

    class Processor ⑥
            implements PropertyMappingAnnotationProcessor<MultiLanguageField> { ⑦
        @Override
        public void process(PropertyMappingStep mapping, MultiLanguageField annotation,
                PropertyMappingAnnotationProcessorContext context) {
            LanguageAlternativeBinderDelegate delegate = new LanguageAlternativeBinderDelegate( ⑧
                    annotation.name().isEmpty() ? null : annotation.name()
            );
            mapping.hostingType() ⑨
                    .binder( AlternativeBinder.create( ⑩
                            Language.class, ⑪
                            context.annotatedElement().name(), ⑫
                            String.class, ⑬
                            BeanReference.ofInstance( delegate ) ⑭
                    ) );
        }
    }
}
```

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

② Allow the annotation to target either methods (getters) or fields.

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

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

⑤ Optionally, define parameters. Here we allow to customize the field name (which will default to the property name, see further down).

⑥ Here the processor class is nested in the annotation class, because it is more convenient, but you are obviously free to implement it in a separate Java file.

⑦ The processor must implement the `PropertyMappingAnnotationProcessor` interface, setting its generic type argument to the type of the corresponding annotation.

⑧ In the annotation processor, instantiate a custom binder delegate (see below for the implementation).

⑨ Access the mapping of the type hosting the property (in this example, `BlogEntry`).

⑩ Apply the `AlternativeBinder` to the type hosting the property (in this example, `BlogEntry`).

⑪ Pass to `AlternativeBinder` the expected type of discriminator values.

⑫ Pass to `AlternativeBinder` the name of the property from which field values should be extracted (in this example, `text`).

⑬ Pass to `AlternativeBinder` the expected type of the property from which index field values are extracted.

⑭ Pass to `AlternativeBinder` the binder delegate.

```
public class LanguageAlternativeBinderDelegate implements AlternativeBinderDelegate<Language, String> { ①

    private final String name;
```

```
        public LanguageAlternativeBinderDelegate(String name) { ②
            this.name = name;
        }

        @Override
        public AlternativeValueBridge<Language, String> bind(IndexSchemaElement indexSchemaElement, ③
                PojoModelProperty fieldValueSource) {
            EnumMap<Language, IndexFieldReference<String>> fields = new EnumMap<>( Language.class );
            String fieldNamePrefix = ( name != null ? name : fieldValueSource.name() ) + "_";

            for ( Language language : Language.values() ) { ④
                String languageCode = language.code;
                IndexFieldReference<String> field = indexSchemaElement.field(
                        fieldNamePrefix + languageCode, ⑤
                        f -> f.asString().analyzer( "text_" + languageCode ) ⑥
                )
                        .toReference();
                fields.put( language, field );
            }

            return new Bridge( fields ); ⑦
        }

        private static class Bridge ⑧
                implements AlternativeValueBridge<Language, String> { ⑨
            private final EnumMap<Language, IndexFieldReference<String>> fields;

            private Bridge(EnumMap<Language, IndexFieldReference<String>> fields) {
                this.fields = fields;
            }

            @Override
            public void write(DocumentElement target, Language discriminator, String bridgedElement) {
                target.addValue( fields.get( discriminator ), bridgedElement ); ⑩
            }
        }
    }
```

① The binder delegate must implement `AlternativeBinderDelegate`. The first type parameter is the expected type of discriminator values (in this example, `Language`); the second type parameter is the expected type of the property from which field values are extracted (in this example, `String`).

② Any (custom) parameter can be passed through the constructor.

③ Implement `bind`, to bind a property to index fields.

④ Define one field per language.

⑤ Make sure to give a different name to each field. Here we're using the language code as a suffix, i.e. `text_en`, `text_fr`, `text_de`, …

⑥ Assign a different analyzer to each field. The analyzers `text_en`, `text_fr`, `text_de` must have been defined in the backend; see Analysis.

⑦ Return a bridge.

⑧ Here the bridge class is nested in the binder class, because it is more convenient, but you are obviously free to implement it as you wish: as a lambda expression, in a separate Java file…

⑨ The bridge must implement the `AlternativeValueBridge` interface.

⑩ The bridge is called when indexing; it selects the field to write to based on the discriminator value, then writes the value to index to that field.

10.9.2. Programmatic mapping

You can apply `AlternativeBinder` through the programmatic mapping too. Behavior and options are identical to annotation-based mapping.

Example 10.45: Applying an `AlternativeBinder` with `.binder(…)`

```
TypeMappingStep blogEntryMapping = mapping.type( BlogEntry.class );
blogEntryMapping.indexed();
blogEntryMapping.property( "language" )
        .marker( AlternativeBinder.alternativeDiscriminator() );
```

```
LanguageAlternativeBinderDelegate delegate = new LanguageAlternativeBinderDelegate( null );
blogEntryMapping.binder( AlternativeBinder.create( Language.class,
        "text", String.class, BeanReference.ofInstance( delegate ) ) );
```

10.10. Tuning when to trigger reindexing

10.10.1. Basics

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

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

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

When the mapping crosses the entity boundaries, things get more complicated. Let's consider a mapping where a `Book` entity is mapped to a document, and that document must include the `name` property of the `Author` entity (for example using `@IndexedEmbedded`). Whenever an author's name changes, Hibernate Search will need to *retrieve all the books of that author*, to reindex them.

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

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

In some rare cases, with the more complex mappings, it is possible that even Hibernate ORM is not aware that an association is bidirectional, because `mappedBy` cannot be used, or because the Standalone POJO Mapper is being used. A few solutions exist:

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

10.10.2. Enriching the entity model with `@AssociationInverseSide`

Given an association from an entity type `A` to entity type `B`, `@AssociationInverseSide` defines the inverse side of an association, i.e. the path from `B` to `A`.

This is mostly useful when using the Standalone POJO Mapper or when using the Hibernate ORM integration and a bidirectional association is not mapped as such in Hibernate ORM (no `mappedBy`).

Example 10.46: Mapping the inverse side of an association with `@AssociationInverseSide`

```
@Entity
@Indexed
public class Book {

    @Id
    @GeneratedValue
    private Integer id;

    private String title;

    @ElementCollection ①
    @JoinTable(
            name = "book_editionbyprice",
            joinColumns = @JoinColumn(name = "book_id")
    )
    @MapKeyJoinColumn(name = "edition_id")
    @Column(name = "price")
    @OrderBy("edition_id asc")
```

```
    @IndexedEmbedded( ②
            name = "editionsForSale",
            extraction = @ContainerExtraction(BuiltinContainerExtractors.MAP_KEY)
    )
    @AssociationInverseSide( ③
            extraction = @ContainerExtraction(BuiltinContainerExtractors.MAP_KEY),
            inversePath = @ObjectPath(@PropertyValue(propertyName = "book"))
    )
    private Map<BookEdition, BigDecimal> priceByEdition = new LinkedHashMap<>();

    public Book() {
    }

    // Getters and setters
    // ...

}
```

```
@Entity
public class BookEdition {

    @Id
    @GeneratedValue
    private Integer id;

    @ManyToOne ④
    private Book book;

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

    public BookEdition() {
    }

    // Getters and setters
    // ...

}
```

① This annotation and the following ones are the Hibernate ORM mapping for a `Map<BookEdition, BigDecimal>` where the keys are `BookEdition` entities and the values are the price of that edition.

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

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

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

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

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

Annotating the property with `@IndexingDependency(derivedFrom = …)` will give Hibernate Search the information it needs and allow triggering reindexing.

Example 10.47: Mapping a derived value with `@IndexingDependency.derivedFrom`

```
@Entity
@Indexed
public class Book {

    @Id
    @GeneratedValue
    private Integer id;

    private String title;
```

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

    public Book() {
    }

    // Getters and setters
    // ...

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

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

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

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

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

10.10.4. Limiting reindexing of containing entities with `@IndexingDependency`

In some cases, triggering reindexing of entities every time a given property changes is not realistically achievable:

- When an association is massive, for example a single entity instance is indexed-embedded in thousands of other entities.
- When a property mapped to the index is updated very frequently, leading to a very frequent reindexing and unacceptable usage of disks or database.
- Etc.

When that happens, it is possible to tell Hibernate Search to ignore updates to a particular property (and, in the case of `@IndexedEmbedded`, anything beyond that property).

Several options are available to control exactly how updates to a given property affect reindexing. See the sections below for an explanation of each option.

`ReindexOnUpdate.SHALLOW`: limiting reindexing to same-entity updates only

`ReindexOnUpdate.SHALLOW` is most useful when an association is highly asymmetric and therefore unidirectional. Think associations to "reference" data such as categories, types, cities, countries, …

It essentially tells Hibernate Search that changing an association — adding or removing associated elements, i.e. "shallow" updates — should trigger reindexing of the object on which the change happened, but changing properties of associated entities — "deep" updates — should not.

For example, let's consider the (incorrect) mapping below:

Example 10.48: A highly-asymmetric, unidirectional association

```
@Entity
@Indexed
public class Book {

    @Id
    private Integer id;

    private String title;

    @ManyToOne ①
    @IndexedEmbedded ②
    private BookCategory category;
```

```
    public Book() {
    }

    // Getters and setters
    // ...

}
```

```
@Entity
public class BookCategory {

    @Id
    private Integer id;

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

③

    // Getters and setters
    // ...

}
```

① Each book has an association to a `BookCategory` entity.

② We want to index-embed the `BookCategory` into the `Book` …

③ … but we really don't want to model the (huge) inverse association from `BookCategory` to `Book`: There are potentially thousands of books for each category, so calling a `getBooks()` method would lead to loading thousands of entities into the Hibernate ORM session at once, and would perform badly. Thus, there isn't any `getBooks()` method to list all books in a category.

With this mapping, Hibernate Search will not be able to reindex all books when the category name changes: the getter that would list all books for that category simply doesn't exist. Since Hibernate Search tries to be safe by default, it will reject this mapping and throw an exception at bootstrap, saying it needs an inverse side to the `Book` → `BookCategory` association.

However, in this case, we don't expect the name of a `BookCategory` to change. That's really "reference" data, which changes so rarely that we can conceivably plan ahead such change and reindex all books whenever that happens. So we would really not mind if Hibernate Search just ignored changes to `BookCategory`…

That's what `@IndexingDependency(reindexOnUpdate = ReindexOnUpdate.SHALLOW)` is for: it tells Hibernate Search to ignore the impact of updates to an associated entity. See the modified mapping below:

Example 10.49: Limiting reindexing to same-entity updates with `ReindexOnUpdate.SHALLOW`

```
@Entity
@Indexed
public class Book {

    @Id
    private Integer id;

    private String title;

    @ManyToOne
    @IndexedEmbedded
    @IndexingDependency(reindexOnUpdate = ReindexOnUpdate.SHALLOW) ①
    private BookCategory category;

    public Book() {
    }

    // Getters and setters
    // ...

}
```

① We use `ReindexOnUpdate.SHALLOW` to tell Hibernate Search that `Book` should be re-indexed when it's assigned a new category (`book.setCategory( newCategory )`), but not when properties of its category change (`category.setName( newName )`).

Hibernate Search will accept the mapping above and boot successfully, since the inverse side of the association from `Book` to `BookCategory` is no longer deemed necessary.

Only *shallow* changes to a book's category will trigger reindexing of that book:

- When a book is assigned a new category (`book.setCategory( newCategory )`), Hibernate Search will consider it a "shallow" change, since it only affects the `Book` entity. Thus, Hibernate Search will reindex the book.
- When a category itself changes (`category.setName( newName )`), Hibernate Search will consider it a "deep" change, since it occurs beyond the boundaries of the `Book` entity. Thus, Hibernate Search will **not** reindex books of that category by itself. The index will become slightly out-of-sync, but this can be solved by reindexing `Book` entities, for example every night.

`ReindexOnUpdate.NO`: disabling reindexing caused by updates of a particular property

`ReindexOnUpdate.NO` is most useful for properties that change very frequently and don't need to be up-to-date in the index.

It essentially tells Hibernate Search that changes to that property should not trigger reindexing,

For example, let's consider the mapping below:

Example 10.50: A frequently-changing property

```
@Entity
@Indexed
public class Sensor {

    @Id
    private Integer id;

    @FullTextField
    private String name; ①

    @KeywordField
    private SensorStatus status; ①

    @Column(name = "\"value\"")
    private double value; ②

    @GenericField
    private double rollingAverage; ③

    public Sensor() {
    }

    // Getters and setters
    // ...

}
```

① The sensor name and status get updated very rarely.
② The sensor value gets updated every few milliseconds
③ When the sensor value gets updated, we also update the rolling average over the last few seconds (based on data not shown here).

Updates to the name and status, which are rarely updated, can perfectly well trigger reindexing. But considering there are thousands of sensors, updates to the sensor value cannot reasonably trigger reindexing: reindexing thousands of sensors every few milliseconds probably won't perform well.

In this scenario, however, search on sensor value is not considered critical and indexes don't need to be as fresh. We can accept indexes to lag behind a few minutes when it comes to a sensor value. We can consider setting up a batch process that runs every few seconds to reindex all sensors, either through a mass indexer, using the Jakarta Batch mass indexing job, or explicitly. So we would really not mind if Hibernate Search just ignored changes to sensor values…

That's what `@IndexingDependency(reindexOnUpdate = ReindexOnUpdate.NO)` is for: it tells Hibernate Search to ignore the impact of updates to the `rollingAverage` property. See the modified mapping below:

Example 10.51: Disabling listener-triggered reindexing for a particular property with `ReindexOnUpdate.NO`

```
@Entity
@Indexed
public class Sensor {

    @Id
    private Integer id;

    @FullTextField
    private String name;

    @KeywordField
    private SensorStatus status;

    @Column(name = "\"value\"")
    private double value;

    @GenericField
    @IndexingDependency(reindexOnUpdate = ReindexOnUpdate.NO) ①
    private double rollingAverage;

    public Sensor() {
    }

    // Getters and setters
    // ...

}
```

① We use `ReindexOnUpdate.NO` to tell Hibernate Search that updates to `rollingAverage` should not trigger reindexing.

With this mapping:

- When a sensor is assigned a new name (`sensor.setName( newName )`) or status (`sensor.setStatus( newStatus )`), Hibernate Search will trigger reindexing of the sensor.
- When a sensor is assigned a new rolling average (`sensor.setRollingAverage( newName )`), Hibernate Search will **not** trigger reindexing of the sensor.

10.10.5. Programmatic mapping

You can control reindexing through the programmatic mapping too. Behavior and options are identical to annotation-based mapping.

Example 10.52: Mapping the inverse side of an association with `.associationInverseSide(…)`

```
TypeMappingStep bookMapping = mapping.type( Book.class );
bookMapping.indexed();
bookMapping.property( "priceByEdition" )
        .indexedEmbedded( "editionsForSale" )
                .extractor( BuiltinContainerExtractors.MAP_KEY )
        .associationInverseSide( PojoModelPath.parse( "book" ) )
                .extractor( BuiltinContainerExtractors.MAP_KEY );
TypeMappingStep bookEditionMapping = mapping.type( BookEdition.class );
bookEditionMapping.property( "label" )
        .fullTextField().analyzer( "english" );
```

Example 10.53: Mapping a derived value with `.indexingDependency().derivedFrom(…)`

```
TypeMappingStep bookMapping = mapping.type( Book.class );
bookMapping.indexed();
bookMapping.property( "mainAuthor" )
        .fullTextField().analyzer( "name" )
        .indexingDependency().derivedFrom( PojoModelPath.parse( "authors" ) );
```

Example 10.54: Limiting triggering reindexing with `.indexingDependency().reindexOnUpdate(…)`

```
TypeMappingStep bookMapping = mapping.type( Book.class );
bookMapping.indexed();
bookMapping.property( "category" )
        .indexedEmbedded()
        .indexingDependency().reindexOnUpdate( ReindexOnUpdate.SHALLOW );
TypeMappingStep bookCategoryMapping = mapping.type( BookCategory.class );
bookCategoryMapping.property( "name" )
        .fullTextField().analyzer( "english" );
```

10.11. Changing the mapping of an existing application

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

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

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

Technically, dropping the index and reindexing is not *strictly* required if the mapping changes include *only*:

- **adding** new indexed entities that will not have any persisted instance, e.g. adding an `@Indexed` annotation on an entity which has no rows in database.
- **adding** new fields that will be empty for all currently persisted entities, e.g. adding a new property on an entity type and mapping it to a field, but with the guarantee that this property will initially be null for every instance of this entity;
- and/or **removing** data from existing indexes/fields, e.g. removing an index field, or removing the need for a field to be stored.

However, you will still need to:

- create missing indexes: this can generally be done automatically by starting up the application with the `create`, `create-or-validate`, or `create-or-update` schema management strategy.
- (Elasticsearch only:) update the schema of existing indexes to declare the new fields. This will be more complex: either do it manually using Elasticsearch's REST API, or start up the application with the `create-or-update` strategy, but be warned that it may fail.

10.12. Custom mapping annotations

10.12.1. Basics

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

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

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

Once this is done, Hibernate Search will be able to detect custom annotations in **indexed classes** (though not necessarily in custom projection types, see Custom root mapping annotations). Whenever a custom annotation is encountered, Hibernate Search will instantiate the annotation processor and call its `process` method, passing the following as arguments:

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

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

- Passing parameters to a value binder/bridge through a custom annotation
- Passing parameters to a property binder/bridge through a custom annotation
- Passing parameters to a type binder/bridge through a custom annotation
- Passing parameters to an identifier binder/bridge through a custom annotation
- Passing parameters to a projection binder through a custom annotation

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

10.12.2. Custom root mapping annotations

To have Hibernate Search consider a custom annotation as a root mapping annotation, add the `@RootMapping` meta-annotation to the custom annotation.

This will ensure Hibernate Search processes annotations on types annotated with the custom annotation even if those types are not referenced in the index mapping, which is mainly useful for custom annotations related to projection mapping.

10.13. Inspecting the mapping

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

Example 10.55: Accessing indexed entities

```
SearchMapping mapping = /* ... */ ①
SearchIndexedEntity<Book> bookEntity = mapping.indexedEntity( Book.class ); ②
String jpaName = bookEntity.jpaName(); ③
IndexManager indexManager = bookEntity.indexManager(); ④
Backend backend = indexManager.backend(); ⑤

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

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

① Retrieve the `SearchMapping`.
② Retrieve the `SearchIndexedEntity` by its entity class. `SearchIndexedEntity` gives access to information pertaining to that entity and its index.
③ (With the Hibernate ORM integration only) Get the JPA name of that entity.
④ Get the index manager for that entity.
⑤ Get the backend for that index manager.
⑥ Retrieve the `SearchIndexedEntity` by its entity name.
⑦ Retrieve all indexed entities.

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

Example 10.56: Accessing the index metamodel

```
SearchIndexedEntity<Book> bookEntity = mapping.indexedEntity( Book.class ); ①
IndexManager indexManager = bookEntity.indexManager(); ②
IndexDescriptor indexDescriptor = indexManager.descriptor(); ③

indexDescriptor.field( "releaseDate" ).ifPresent( field -> { ④
    String path = field.absolutePath(); ⑤
    String relativeName = field.relativeName();
    // Etc.

    if ( field.isValueField() ) { ⑥
        IndexValueFieldDescriptor valueField = field.toValueField(); ⑦
```

```
            IndexValueFieldTypeDescriptor type = valueField.type(); ⑧
            boolean projectable = type.projectable();
            Class<?> dslArgumentClass = type.dslArgumentClass();
            Class<?> projectedValueClass = type.projectedValueClass();
            Optional<String> analyzerName = type.analyzerName();
            Optional<String> searchAnalyzerName = type.searchAnalyzerName();
            Optional<String> normalizerName = type.normalizerName();
            // Etc.
            Set<String> traits = type.traits(); ⑨
            if ( traits.contains( IndexFieldTraits.Aggregations.RANGE ) ) {
                // ...
            }
        }
        else if ( field.isObjectField() ) { ⑩
            IndexObjectFieldDescriptor objectField = field.toObjectField();

            IndexObjectFieldTypeDescriptor type = objectField.type();
            boolean nested = type.nested();
            // Etc.
        }
    } );

    Collection<? extends AnalyzerDescriptor> analyzerDescriptors = indexDescriptor.analyzers(); ⑪
    for ( AnalyzerDescriptor analyzerDescriptor : analyzerDescriptors ) {
        String analyzerName = analyzerDescriptor.name();
        // ...
    }

    Optional<? extends AnalyzerDescriptor> analyzerDescriptor = indexDescriptor.analyzer( "some-analyzer-name" ); ⑫
    // ...

    Collection<? extends NormalizerDescriptor> normalizerDescriptors = indexDescriptor.normalizers(); ⑬
    for ( NormalizerDescriptor normalizerDescriptor : normalizerDescriptors ) {
        String normalizerName = normalizerDescriptor.name();
        // ...
    }

    Optional<? extends NormalizerDescriptor> normalizerDescriptor = indexDescriptor.normalizer( "some-normalizer-
    name" ); ⑭
    // ...
```

① Retrieve a `SearchIndexedEntity`.

② Get the index manager for that entity. `IndexManager` gives access to information pertaining to the index. This includes the metamodel, but not only (see below).

③ Get the descriptor for that index. The descriptor exposes the index metamodel.

④ Retrieve a field by name. The method returns an `Optional`, which is empty if the field does not exist.

⑤ The field descriptor exposes information about the field structure: path, name, parent, …

⑥ Check that the field is a value field, holding a value (integer, text, …), as opposed to object fields, holding other fields.

⑦ Narrow down the field descriptor to a value field descriptor.

⑧ Get the descriptor for the field type. The type descriptor exposes information about the field's capabilities: is it searchable, sortable, projectable, what is the expected java class for arguments to the Search DSL, what are the analyzers/normalizer set on this field, …

⑨ Inspect the "traits" of a field type.

⑩ Object fields can also be inspected.

⑪ A collection of all configured analyzers available for the index represented by the descriptor can also be inspected.

⑫ Alternatively, analyzer descriptors can be retrieved by name to see if a particular analyzer is available within the index context.

⑬ A collection of all configured normalizers available for the index represented by the descriptor can also be inspected.

⑭ Alternatively, normalizer descriptors can be retrieved by name to see if a particular normalizer is available within the index context.

Traits define the set of search capabilities available for the field, in particular, each trait represents a predicate/sort/projection/aggregation that can be used on that field.

The `Backend` and `IndexManager` can also be used to retrieve the Elasticsearch REST client or retrieve Lucene analyzers.

The `SearchMapping` also exposes methods to retrieve an `IndexManager` by name, or even a whole `Backend` by name.

Chapter 11. Mapping index content to custom types (projection constructors)

11.1. Basics

Projections allow retrieving data directly from matched documents as the result of a search query. As the structure of documents and projections becomes more complex, so do programmatic calls to the Projection DSL, which can lead to overwhelming projection definitions.

To address this, Hibernate Search offers the ability to define projections through the mapping of custom types (typically records), by applying the `@ProjectionConstructor` annotation to those types or their constructor. Executing such a projection then becomes as easy as referencing the custom type.

Such projections are composite, their inner projections (components) being inferred from the name and type of the projection constructors' parameters.

There are a few constraints to keep in mind when annotating a custom projection type:

- The custom projection type must be in the same JAR as entity types, or Hibernate Search will require additional configuration.
- When projecting on value fields or object fields, the path to the projected field is inferred from the constructor parameter name by default, but inference will fail if constructor parameter names are not included in the Java bytecode. Alternatively the path can be provided explicitly through `@FieldProjection(path = …)` /`@ObjectProjection(path = …)`, in which case Hibernate Search won't rely on constructor parameter names.
- When projecting on value fields, the constraints of the `field` projection still apply. In particular, with the Lucene backend, value fields involved in the projection must be configured as projectable.
- When projecting on object fields, the constraints of the `object` projection still apply. In particular, with the Lucene backend, multi-valued object fields involved in the projection must be configured as nested.

Example 11.1: Using a custom record type to project data from the index

```
@ProjectionConstructor ①
public record MyBookProjection(
        @IdProjection Integer id, ②
        String title, ③
        List<Author> authors) { ④
    @ProjectionConstructor ⑤
    public record Author(String firstName, String lastName) {
    }
}
```

① Annotate the record type with `@ProjectionConstructor`, either at the type level (if there's only one constructor) or at the constructor level (if there are multiple constructors).

② To project on the entity identifier, annotate the relevant constructor parameter with `@IdProjection`.

Most projections have a corresponding annotation that can be used on constructor parameters.

③ To project on a value field, add a constructor parameter named after that field and with the same type as that field. See Implicit inner projection inference for more information on how constructor parameters should be defined.

Alternatively, the field projection can be configured explicitly with `@FieldProjection`.

④ To project on an object field, add a constructor parameter named after that field and with its own custom projection type. Multivalued projections must be modeled as one of the multivalued containers for which a collector is available in `ProjectionCollector` or their supertype.

Alternatively, the object projection can be configured explicitly with `@ObjectProjection`.

⑤ Annotate any custom projection type used for object fields with `@ProjectionConstructor` as well.

```
List<MyBookProjection> hits = searchSession.search( Book.class )
        .select( MyBookProjection.class ) ①
        .where( f -> f.matchAll() )
        .fetchHits( 20 ); ②
```

① Pass the custom projection type to `.select(…)`.

② Each hit will be an instance of the custom projection type, populated with data retrieved from the index.

Custom, non-record classes can also be annotated with `@ProjectionConstructor`, which can be useful if you cannot use records for some reason (for example because you're still using Java 13 or below).

The example above executes a projection equivalent to the following code:

Example 11.2: Programmatic projection definition equivalent to the previous example

```
List<MyBookProjection> hits = searchSession.search( Book.class )
        .select( f -> f.composite()
                .from(
                        f.id( Integer.class ),
                        f.field( "title", String.class ),
                        f.object( "authors" )
                                .from(
                                        f.field( "authors.firstName", String.class ),
                                        f.field( "authors.lastName", String.class )
                                )
                                .as( MyBookProjection.Author::new )
                                .collector( ProjectionCollector.list() )
                )
                .as( MyBookProjection::new ) )
        .where( f -> f.matchAll() )
        .fetchHits( 20 );
```

11.2. Detection of mapped projection types

Hibernate Search must know of projection types on startup, which it generally does as soon as they are annotated with `@ProjectionConstructor`, thanks to classpath scanning.

For more information about classpath scanning and how to tune it (for example to scan dependencies instead of just the application JAR), see Classpath scanning.

11.3. Implicit inner projection inference

11.3.1. Basics

When constructor parameters are not annotated with explicit projection annotations, Hibernate Search applies some basic inference rules based on the name and type of those parameters in order to select (inner) projections.

The following sections explain how to define the name and type of constructor parameters to get the desired projection.

11.3.2. Inner projection and type

When a constructor parameter is not annotated with an explicit projection annotation, Hibernate Search infers the type of the inner projection from the type of the corresponding constructor parameter.

You should set the type of a constructor parameter according to the following rules:

- For a single-valued projection:
  - For a projection on a value field (generally mapped using `@FullTextField`/`@GenericField`/etc.), set the parameter type to the type of projected values for the target field, which in general is the type of the property annotated with `@FullTextField`/`@GenericField`/etc.
  - For a projection on an object field (generally mapped using `@IndexedEmbedded`), set the parameter type to another custom type annotated with `@ProjectionConstructor`, whose constructor will define which fields to extract from that object field.
- For projections where values are wrapped in a container, be it a multivalued projection represented by some collection or array, or a single-valued projection wrapped in an optional, follow the rules above for the elements inside the container and then wrap the type with one of the containers available in `ProjectionCollector` (`Iterable`, `Collection`, `List`, etc.), e.g. `Iterable<SomeType>`, `Collection<SomeType>`, `List<SomeType>`, etc.

Constructor parameters meant to represent a multivalued projection **must** have the type of one of the supported multivalued containers.

In case the `ProjectionCollector` does not provide a suitable collector for a container/collection needed for a constructor parameter mapping, a custom projection binding can be implemented and a user-implemented projection collector

applied.

11.3.3. Inner projection and field path

When a constructor parameter is not annotated with an explicit projection annotation or when it is but that annotation does not provide an explicit path, Hibernate Search infers the path of the field to project on from the name of the corresponding constructor parameter.

In that case, you should set the name of a constructor parameter (in the Java code) to the name of the field to project on.

Hibernate Search can only retrieve the name of the constructor parameter:

- For the canonical constructor of record types, regardless of compiler flags.
- For constructors of non-record types or non-canonical constructors of record types if and only if the type was compiled with the `-parameters` compiler flag.

11.4. Explicit inner projection

Constructor parameters can be annotated with explicit projection annotations such as `@IdProjection` or `@FieldProjection`.

For projections that would normally be inferred automatically, this allows further customization, for example in a field projection to set the target field path explicitly or to disable value conversion. Alternatively, in an object projection, this also allows breaking cycles of nested object projections.

For other projections such as identifier projection, this is actually the only way to use them in a projection constructor, because they would never be inferred automatically.

See the documentation of each projection for more information about the corresponding built-in annotation to be applied to projection constructor parameters.

11.5. Mapping types with multiple constructors

If the projection type (record or class) has multiple constructors, the `@ProjectionConstructor` annotation cannot be applied at the type level and must be applied to the constructor you wish to use for projections.

Example 11.3: Annotating a specific constructor with `@ProjectionConstructor`

```
public class MyAuthorProjectionClassMultiConstructor {
    public final String firstName;
    public final String lastName;

    @ProjectionConstructor ①
    public MyAuthorProjectionClassMultiConstructor(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public MyAuthorProjectionClassMultiConstructor(String fullName) { ②
        this( fullName.split( " " )[0], fullName.split( " " )[1] );
    }

    // ... Equals and hashcode ...

}
```

① Annotate the constructor to use for projections with `@ProjectionConstructor`.
② Other constructors can be used for other purposes than projections, but they **must not** be annotated with `@ProjectionConstructor` (only one such constructor is allowed).

In the case of records, the (implicit) canonical constructor can also be annotated, but it requires representing that constructor in the code with a specific syntax:

Example 11.4: Annotating the canonical constructor with `@ProjectionConstructor`

```
public record MyAuthorProjectionRecordMultiConstructor(String firstName, String lastName) {
    @ProjectionConstructor ①
```

```
        public MyAuthorProjectionRecordMultiConstructor { ②
        }

        public MyAuthorProjectionRecordMultiConstructor(String fullName) { ③
            this( fullName.split( " " )[0], fullName.split( " " )[1] );
        }
    }
```

① Annotate the constructor to use for projections with `@ProjectionConstructor`.
② The (implicit) canonical constructor uses a specific syntax, without parentheses or parameters.
③ Other constructors can be used for other purposes than projections, but they **must not** be annotated with `@ProjectionConstructor` (only one such constructor is allowed).

11.6. Programmatic mapping

You can map projection constructors through the programmatic mapping too. Behavior and options are identical to annotation-based mapping.

Example 11.5: Mapping the main projection constructor with `.projectionConstructor()` and `.projection(<binder>)`

```
TypeMappingStep myBookProjectionMapping = mapping.type( MyBookProjection.class );
myBookProjectionMapping.mainConstructor()
        .projectionConstructor(); ①
myBookProjectionMapping.mainConstructor().parameter( 0 )
        .projection( IdProjectionBinder.create() ); ②
TypeMappingStep myAuthorProjectionMapping = mapping.type( MyBookProjection.Author.class );
myAuthorProjectionMapping.mainConstructor()
        .projectionConstructor();
```

① Mark the constructor as a projection constructor.
② The equivalent to explicit projection annotations is to pass projection binder instances: there is a built-in projection binder for every built-in projection annotation.

If the projection type (record or class) has multiple constructors, you will need to use `.constructor(…)` instead of `.mainConstructor()`, passing the (raw) type of the constructor parameters as arguments.

Example 11.6: Mapping a specific projection constructor with `.projectionConstructor()`

```
mapping.type( MyAuthorProjectionClassMultiConstructor.class )
        .constructor( String.class, String.class )
        .projectionConstructor();
```

Chapter 12. Binding and bridges

12.1. Basics

In Hibernate Search, binding is the process of assigning custom components to the domain model.

The most intuitive components that can be bound are bridges, responsible for converting pieces of data from the entity model to the document model.

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

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

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

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

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

Table 12.1: Comparison of available bridge types

Bridge type	`ValueBridge`	`PropertyBridge`	`TypeBridge`	`IdentifierBridge`	`RoutingBridge`
Applied to…	Class field or getter	Class field or getter	Class	Class field or getter (usually entity ID)	Class
Maps to…	One index field. Value field only: integer, text, geopoint, etc. No object field (composite).	One index field or more. Value fields as well as object fields (composite).	One index field or more. Value fields as well as object fields (composite).	Document identifier	Route (conditional indexing, routing key)
Built-in annotation(s)	`@GenericField`, `@FullTextField`, …	`@PropertyBinding`	`@TypeBinding`	`@DocumentId`	`@Indexed( routingBinder = … )`
Supports container extractors	Yes	No	No	No	No
Supports mutable types	**No**	Yes	Yes	**No**	Yes

Not all binders are about indexing, however. The constructor parameters involved in projection constructors can be bound as well; you will find more information about that in this section.

12.2. Value bridge

12.2.1. Basics

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

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

- The type of the index field can be customized directly in the `@*Field` annotation: it can be defined as sortable, projectable, it can be assigned an analyzer, …
- The bridge can be transparently applied to elements of a container. For example, you can implement a `ValueBridge<ISBN, String>` and transparently use it on a property of type `List<ISBN>`: the bridge will simply be applied once per list element and populate the index field with as many values.

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

- A value bridge only allows one-to-one mapping: one property to one index field. A single value bridge cannot populate more than one index field.
- A value bridge **will not work correctly when applied to a mutable type**. A value bridge is expected to be applied to "atomic" data, such as a `LocalDate`; if it is applied to an entity, for example, extracting data from its properties, Hibernate Search will not be aware of which properties are used and will not be able to detect that reindexing is required when these properties change.

Below is an example of a custom value bridge that converts a custom `ISBN` type to its string representation to index it:

Example 12.1: Implementing and using a `ValueBridge`

```
public class ISBNValueBridge implements ValueBridge<ISBN, String> { ①

    @Override
    public String toIndexedValue(ISBN value, ValueBridgeToIndexedValueContext context) { ②
        return value == null ? null : value.getStringValue();
    }

}
```

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

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

```
@Entity
@Indexed
public class Book {

    @Id
    @GeneratedValue
    private Integer id;

    @Convert(converter = ISBNAttributeConverter.class) ①
    @KeywordField( ②
            valueBridge = @ValueBridgeRef(type = ISBNValueBridge.class), ③
            normalizer = "isbn" ④
    )
    private ISBN isbn;

    // Getters and setters
    // ...

}
```

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

② Map the property to an index field.

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

④ Customize the field as usual.

Here is an example of what an indexed document would look like, with the Elasticsearch backend:

```
{
  "isbn": "978-0-58-600835-5"
}
```

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

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

See the next sections for more information.

12.2.2. Type resolution

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

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

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

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

12.2.3. Using value bridges in other `@*Field` annotations

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

For example:

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

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

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

12.2.4. Supporting projections with `fromIndexedValue()`

By default, any attempt to project on a field using a custom bridge will result in an exception, because Hibernate Search doesn't know how to convert the projected values obtained from the index back to the property type.

It is possible to disable conversion explicitly to get the raw value from the index, but another way of solving the problem is to simply implement `fromIndexedValue` in the custom bridge. This method will be called whenever a projected value needs to be converted.

Example 12.2: Implementing `fromIndexedValue` to convert projected values

```
public class ISBNValueBridge implements ValueBridge<ISBN, String> {

    @Override
    public String toIndexedValue(ISBN value, ValueBridgeToIndexedValueContext context) {
        return value == null ? null : value.getStringValue();
    }

    @Override
    public ISBN fromIndexedValue(String value, ValueBridgeFromIndexedValueContext context) {
        return value == null ? null : ISBN.parse( value ); ①
    }
}
```

① Implement `fromIndexedValue` as necessary.

```
@Entity
@Indexed
public class Book {

    @Id
    @GeneratedValue
    private Integer id;

    @Convert(converter = ISBNAttributeConverter.class) ①
    @KeywordField( ②
            valueBridge = @ValueBridgeRef(type = ISBNValueBridge.class), ③
            normalizer = "isbn",
            projectable = Projectable.YES ④
    )
    private ISBN isbn;
```

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

}
```

① This is unrelated to the value bridge, but necessary in order for Hibernate ORM to store the data correctly in the database.
② Map the property to an index field.
③ Instruct Hibernate Search to use our custom value bridge.
④ Do not forget to configure the field as projectable.

12.2.5. Parsing the string representation to an index field type with `parse()`

By default, when a custom bridge is used, some Hibernate Search features like specifying the `indexNullAs` attribute of `@*Field` annotations, or using a field with such a custom bridge in query string predicates (`simpleQueryString()`/`queryString()`) with local backends (e.g. Lucene), or when using the `ValueModel.STRING` in the Search DSL will not work out of the box.

In order to make it work, the bridge needs to implement the `parse` method so that Hibernate Search can convert the string representation to a value of the correct type for the index field.

Example 12.3: Implementing `parse`

```
public class ISBNValueBridge implements ValueBridge<ISBN, String> {

    @Override
    public String toIndexedValue(ISBN value, ValueBridgeToIndexedValueContext context) {
        return value == null ? null : value.getStringValue();
    }

    @Override
    public String parse(String value) {
        // Just check the string format and return the string
        return ISBN.parse( value ).getStringValue(); ①
    }
}
```

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

```
@Entity
@Indexed
public class Book {

    @Id
    @GeneratedValue
    private Integer id;

    @Convert(converter = ISBNAttributeConverter.class) ①
    @KeywordField( ②
            valueBridge = @ValueBridgeRef(type = ISBNValueBridge.class), ③
            normalizer = "isbn",
            indexNullAs = "000-0-00-000000-0" ④
    )
    private ISBN isbn;

    // Getters and setters
    // ...

}
```

① This is unrelated to the value bridge, but necessary in order for Hibernate ORM to store the data correctly in the database.
② Map the property to an index field.
③ Instruct Hibernate Search to use our custom value bridge.
④ Set `indexNullAs` to a valid value.

```
List<Book> result = searchSession.search( Book.class )
        .where( f -> f.queryString().field( "isbn" )
```

```
                .matching( "978-0-13-468599-1" ) ) ①
        .fetchHits( 20 );
```

① Use a string representation of an ISBN in a query string predicate.

12.2.6. Formatting the value as string with `format()`

By default, when a custom bridge is used, requesting a `ValueModel.STRING` for a field projection will use a simple `toString()` call.

In order to customize the format, the bridge needs to implement the `format` method so that Hibernate Search can convert the index field to the desired string representation.

Example 12.4: Implementing `format`

```
public class ISBNValueBridge implements ValueBridge<ISBN, Long> {

    // Implement mandatory toDocumentIdentifier/fromDocumentIdentifier ...
    // ...

    @Override
    public String format(Long value) { ①
        return value == null
                ? null
                : value.toString()
                        .replaceAll( "(\\d{3})(\\d)(\\d{2})(\\d{6})(\\d)", "$1-$2-$3-$4-$5" );
    }
}
```

① Implement `format` as necessary. The bridge may throw exceptions for invalid values.

```
@Entity
@Indexed
public class Book {

    @Id
    @GeneratedValue
    private Integer id;

    @Convert(converter = ISBNAttributeConverter.class) ①
    @GenericField( ②
            valueBridge = @ValueBridgeRef(type = ISBNValueBridge.class), ③
            projectable = Projectable.YES ④
    )
    private ISBN isbn;

    // Getters and setters
    // ...

}
```

① This is unrelated to the value bridge, but necessary in order for Hibernate ORM to store the data correctly in the database.
② Map the property to an index field.
③ Instruct Hibernate Search to use our custom value bridge.
④ Configure the field as projectable.

```
List<String> result = searchSession.search( Book.class )
        .select( f -> f.field( "isbn", String.class, ValueModel.STRING ) ) ①
        .where( f -> f.matchAll() )
        .fetchHits( 20 );
```

① Use a string representation when requesting the field projection.

12.2.7. Compatibility across indexes with `isCompatibleWith()`

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

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

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

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

Example 12.5: Implementing `isCompatibleWith` to support multi-index search

```
public class ISBNValueBridge implements ValueBridge<ISBN, String> {

    @Override
    public String toIndexedValue(ISBN value, ValueBridgeToIndexedValueContext context) {
        return value == null ? null : value.getStringValue();
    }

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

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

12.2.8. Configuring the bridge more finely with `ValueBinder`

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

Example 12.6: Implementing a `ValueBinder`

```
public class ISBNValueBinder implements ValueBinder { ①
    @Override
    public void bind(ValueBindingContext<?> context) { ②
        context.bridge( ③
                ISBN.class, ④
                new ISBNValueBridge(), ⑤
                context.typeFactory() ⑥
                        .asString() ⑦
                        .normalizer( "isbn" ) ⑧
        );
    }

    private static class ISBNValueBridge implements ValueBridge<ISBN, String> {
        @Override
        public String toIndexedValue(ISBN value, ValueBridgeToIndexedValueContext context) {
            return value == null ? null : value.getStringValue(); ⑨
        }
    }
}
```

① The binder must implement the `ValueBinder` interface.
② Implement the `bind` method.
③ Call `context.bridge(…)` to define the value bridge to use.
④ Pass the expected type of property values.
⑤ Pass the value bridge instance.
⑥ Use the context's type factory to create an index field type.
⑦ Pick a base type for the index field using an `as*()` method.

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

⑨ The value bridge must still be implemented.
Here the bridge class is nested in the binder class, because it is more convenient, but you are obviously free to implement it as you wish: as a lambda expression, in a separate Java file…

```
@Entity
@Indexed
public class Book {

    @Id
    @GeneratedValue
    private Integer id;

    @Convert(converter = ISBNAttributeConverter.class) ①
    @KeywordField( ②
            valueBinder = @ValueBinderRef(type = ISBNValueBinder.class), ③
            sortable = Sortable.YES ④
    )
    private ISBN isbn;

    // Getters and setters
    // ...

}
```

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

② Map the property to an index field.

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

④ Customize the field as usual. Configuration set using annotation attributes take precedence over the index field type configuration set by the value binder. For example, in this case, the field with be sortable even if the binder didn't define the field as sortable.

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

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

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

12.2.9. Passing parameters

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

However, these parameters are not passed to the value bridge or value binder. There are two ways to pass parameters to value bridges:

- One is (mostly) limited to string parameters, but is trivial to implement.
- The other can allow any type of parameters, but requires you to declare your own annotations.

Simple, string parameters

You can define string parameters to the `@ValueBinderRef` annotation and then use them later in the binder:

Example 12.7: Passing parameters to a `ValueBridge` using the `@ValueBinderRef` annotation

```
public class BooleanAsStringBridge implements ValueBridge<Boolean, String> { ①

    private final String trueAsString;
    private final String falseAsString;

    public BooleanAsStringBridge(String trueAsString, String falseAsString) { ②
```

```
            this.trueAsString = trueAsString;
            this.falseAsString = falseAsString;
        }

        @Override
        public String toIndexedValue(Boolean value, ValueBridgeToIndexedValueContext context) {
            if ( value == null ) {
                return null;
            }
            return value ? trueAsString : falseAsString;
        }
    }
```

① Implement a bridge that does not index booleans directly, but indexes them as strings instead.
② The bridge accepts two parameters in its constructors: the string representing `true` and the string representing `false`.

```
    public class BooleanAsStringBinder implements ValueBinder {

        @Override
        public void bind(ValueBindingContext<?> context) {
            String trueAsString = context.params().get( "trueAsString", String.class ); ①
            String falseAsString = context.params().get( "falseAsString", String.class );

            context.bridge( Boolean.class, ②
                    new BooleanAsStringBridge( trueAsString, falseAsString ) );
        }
    }
```

① Use the binding context to get the parameter values.
The `param` method will throw an exception if the parameter has not been defined. Alternatively, use `paramOptional` to get an `java.util.Optional` that will be empty if the parameter has not been defined.

② Pass them as arguments to the bridge constructor.

```
    @Entity
    @Indexed
    public class Book {

        @Id
        @GeneratedValue
        private Integer id;

        private String title;

        @GenericField(valueBinder = @ValueBinderRef(type = BooleanAsStringBinder.class, ①
                params = {
                        @Param(name = "trueAsString", value = "yes"),
                        @Param(name = "falseAsString", value = "no")
                }))
        private boolean published;

        @ElementCollection
        @GenericField(valueBinder = @ValueBinderRef(type = BooleanAsStringBinder.class, ②
                params = {
                        @Param(name = "trueAsString", value = "passed"),
                        @Param(name = "falseAsString", value = "failed")
                }), name = "censorshipAssessments_allYears")
        private Map<Year, Boolean> censorshipAssessments = new HashMap<>();

        // Getters and setters
        // ...

    }
```

① Define the binder to use on the property, setting the `fieldName` parameter.
② Because we use a value bridge, the annotation can be transparently applied to containers. Here, the bridge will be applied successively to each value in the map.

Parameters with custom annotations

You can pass parameters of any type to the bridge by defining a custom annotation with attributes:

Example 12.8: Passing parameters to a `ValueBridge` using a custom annotation

```
public class BooleanAsStringBridge implements ValueBridge<Boolean, String> { ①

    private final String trueAsString;
    private final String falseAsString;

    public BooleanAsStringBridge(String trueAsString, String falseAsString) { ②
        this.trueAsString = trueAsString;
        this.falseAsString = falseAsString;
    }

    @Override
    public String toIndexedValue(Boolean value, ValueBridgeToIndexedValueContext context) {
        if ( value == null ) {
            return null;
        }
        return value ? trueAsString : falseAsString;
    }
}
```

① Implement a bridge that does not index booleans directly, but indexes them as strings instead.
② The bridge accepts two parameters in its constructors: the string representing `true` and the string representing `false`.

```
@Retention(RetentionPolicy.RUNTIME) ①
@Target({ ElementType.METHOD, ElementType.FIELD }) ②
@PropertyMapping(processor = @PropertyMappingAnnotationProcessorRef( ③
        type = BooleanAsStringField.Processor.class
))
@Documented ④
@Repeatable(BooleanAsStringField.List.class) ⑤
public @interface BooleanAsStringField {

    String trueAsString() default "true"; ⑥

    String falseAsString() default "false";

    String name() default ""; ⑦

    ContainerExtraction extraction() default @ContainerExtraction(); ⑦

    @Documented
    @Target({ ElementType.METHOD, ElementType.FIELD })
    @Retention(RetentionPolicy.RUNTIME)
    @interface List {
        BooleanAsStringField[] value();
    }

    class Processor ⑧
            implements PropertyMappingAnnotationProcessor<BooleanAsStringField> { ⑨
        @Override
        public void process(PropertyMappingStep mapping, BooleanAsStringField annotation,
                PropertyMappingAnnotationProcessorContext context) {
            BooleanAsStringBridge bridge = new BooleanAsStringBridge( ⑩
                    annotation.trueAsString(), annotation.falseAsString()
            );
            mapping.genericField( ⑪
                        annotation.name().isEmpty() ? null : annotation.name()
                    )
                    .valueBridge( bridge ) ⑫
                    .extractors( ⑬
                            context.toContainerExtractorPath( annotation.extraction() )
                    );
        }
    }
```

```
}
```

① Define an annotation with `RUNTIME` retention. **Any other retention policy will cause the annotation to be ignored by Hibernate Search**.
② Since we're defining a value bridge, allow the annotation to target either methods (getters) or fields.
③ Mark this annotation as a property mapping, and instruct Hibernate Search to apply the given processor whenever it finds this annotation. It is also possible to reference the processor by its CDI/Spring bean name.
④ Optionally, mark the annotation as documented, so that it is included in the javadoc of your entities.
⑤ Optionally, mark the annotation as repeatable, in order to be able to declare multiple fields on the same property.
⑥ Define custom attributes to configure the value bridge. Here we define two strings that the bridge should use to represent the boolean values `true` and `false`.
⑦ Since we will be using a custom annotation, and not the built-in `@*Field` annotation, the standard parameters that make sense for this bridge need to be declared here, too.
⑧ Here the processor class is nested in the annotation class, because it is more convenient, but you are obviously free to implement it in a separate Java file.
⑨ The processor must implement the `PropertyMappingAnnotationProcessor` interface, setting its generic type argument to the type of the corresponding annotation.
⑩ In the `process` method, instantiate the bridge and pass the annotation attributes as constructor arguments.
⑪ Declare the field with the configured name (if provided).
⑫ Assign our bridge to the field. Alternatively, we could assign a value binder instead, using the `valueBinder()` method.
⑬ Configure the remaining standard parameters. Note that the `context` object passed to the `process` method exposes utility methods to convert standard Hibernate Search annotations to something that can be passed to the mapping (here, `@ContainerExtraction` is converted to a container extractor path).

```
@Entity
@Indexed
public class Book {

    @Id
    @GeneratedValue
    private Integer id;

    private String title;

    @BooleanAsStringField(trueAsString = "yes", falseAsString = "no") ①
    private boolean published;

    @ElementCollection
    @BooleanAsStringField( ②
            name = "censorshipAssessments_allYears",
            trueAsString = "passed", falseAsString = "failed"
    )
    private Map<Year, Boolean> censorshipAssessments = new HashMap<>();

    // Getters and setters
    // ...

}
```

① Apply the bridge using its custom annotation, setting the parameters.
② Because we use a value bridge, the annotation can be transparently applied to containers. Here, the bridge will be applied successively to each value in the map.

12.2.10. Accessing the ORM session or session factory from the bridge

This feature is only available with the Hibernate ORM integration.

It **cannot** be used with the Standalone POJO Mapper in particular.

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

Example 12.9: Retrieving the ORM session or session factory from a `ValueBridge`

```
public class MyDataValueBridge implements ValueBridge<MyData, String> {

    @Override
    public String toIndexedValue(MyData value, ValueBridgeToIndexedValueContext context) {
        SessionFactory sessionFactory = context.extension( HibernateOrmExtension.get() ) ①
                .sessionFactory(); ②
        // ... do something with the factory ...
    }

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

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

12.2.11. Injecting beans into the value bridge or value binder

With compatible frameworks, Hibernate Search supports injecting beans into both the `ValueBridge` and the `ValueBinder`.

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

The context passed to the value binder's `bind` method also exposes a `beanResolver()` method to access the bean resolver and instantiate beans explicitly.

See Bean injection for more details.

12.2.12. Programmatic mapping

You can apply a value bridge through the programmatic mapping too. Just pass an instance of the bridge.

Example 12.10: Applying a `ValueBridge` with `.valueBridge(…)`

```
TypeMappingStep bookMapping = mapping.type( Book.class );
bookMapping.indexed();
bookMapping.property( "isbn" )
        .keywordField().valueBridge( new ISBNValueBridge() );
```

Similarly, you can pass a binder instance. You can pass arguments either through the binder's constructor or through setters.

Example 12.11: Applying a `ValueBinder` with `.valueBinder(…)`

```
TypeMappingStep bookMapping = mapping.type( Book.class );
bookMapping.indexed();
bookMapping.property( "isbn" )
        .genericField()
                .valueBinder( new ISBNValueBinder() )
                .sortable( Sortable.YES );
```

12.2.13. Incubating features

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

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

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

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

See the javadoc for more information.

12.3. Property bridge

12.3.1. Basics

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

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

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

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

Implementing a property bridge requires two components:

1. A custom implementation of `PropertyBinder`, to bind the bridge to a property at bootstrap. This involves declaring the parts of the property that will be used, declaring the index fields that will be populated along with their type, and instantiating the property bridge.
2. A custom implementation of `PropertyBridge`, to perform the conversion at runtime. This involves extracting data from the property, transforming it if necessary, and pushing it to index fields.

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

Example 12.12: Implementing and using a `PropertyBridge`

```
public class InvoiceLineItemsSummaryBinder implements PropertyBinder { ①

    @Override
    public void bind(PropertyBindingContext context) { ②
        context.dependencies() ③
                .use( "category" )
                .use( "amount" );

        IndexSchemaObjectField summaryField = context.indexSchemaElement() ④
                .objectField( "summary" );

        IndexFieldType<BigDecimal> amountFieldType = context.typeFactory() ⑤
                .asBigDecimal().decimalScale( 2 ).toIndexFieldType();

        context.bridge( ⑥
                List.class, ⑦
                new Bridge( ⑧
                        summaryField.toReference(), ⑨
                        summaryField.field( "total", amountFieldType ).toReference(), ⑩
                        summaryField.field( "books", amountFieldType ).toReference(), ⑩
                        summaryField.field( "shipping", amountFieldType ).toReference() ⑩
                )
        );
    }

    // ... class continues below
```

① The binder must implement the `PropertyBinder` interface.

② Implement the `bind` method in the binder.

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

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

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

⑥ Call `context.bridge(…)` to define the property bridge to use.

⑦ Pass the expected type of property.

⑧ Pass the property bridge instance.

⑨ Pass a reference to the `summary` object field to the bridge.

⑩ Create a subfield for the `total` amount of the invoice, a subfield for the subtotal for `books`, and a subfield for the subtotal for `shipping`. Pass references to these fields to the bridge.

```
        // ... class InvoiceLineItemsSummaryBinder (continued)

        @SuppressWarnings("rawtypes")
        private static class Bridge ①
                implements PropertyBridge<List> { ②

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

            private Bridge(IndexObjectFieldReference summaryField, ③
                    IndexFieldReference<BigDecimal> totalField,
                    IndexFieldReference<BigDecimal> booksField,
                    IndexFieldReference<BigDecimal> shippingField) {
                this.summaryField = summaryField;
                this.totalField = totalField;
                this.booksField = booksField;
                this.shippingField = shippingField;
            }

            @Override
            public void write(DocumentElement target, List bridgedElement, PropertyBridgeWriteContext context) { ④
                @SuppressWarnings("unchecked")
                List<InvoiceLineItem> lineItems = (List<InvoiceLineItem>) bridgedElement;

                BigDecimal total = BigDecimal.ZERO;
                BigDecimal books = BigDecimal.ZERO;
                BigDecimal shipping = BigDecimal.ZERO;
                for ( InvoiceLineItem lineItem : lineItems ) { ⑤
                    BigDecimal amount = lineItem.getAmount();
                    total = total.add( amount );
                    switch ( lineItem.getCategory() ) {
                        case BOOK:
                            books = books.add( amount );
                            break;
                        case SHIPPING:
                            shipping = shipping.add( amount );
                            break;
                    }
                }

                DocumentElement summary = target.addObject( this.summaryField ); ⑥
                summary.addValue( this.totalField, total ); ⑦
                summary.addValue( this.booksField, books ); ⑦
                summary.addValue( this.shippingField, shipping ); ⑦
            }
        }
    }
```

① Here the bridge class is nested in the binder class, because it is more convenient, but you are obviously free to implement it as

you wish: as a lambda expression, in a separate Java file…

② The bridge must implement the `PropertyBridge` interface. One generic type argument must be provided: the type of the property, i.e. the type of the "bridged element".

③ The bridge stores references to the fields — it will need them when indexing.

④ Implement the `write` method in the bridge. This method is called on indexing.

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

⑥ Add an object to the `summary` object field. Note the `summary` field was declared at the root, so we call `addObject` directly on the `target` argument.

⑦ Add a value to each of the `summary.total`, `summary.books` and `summary.shipping` fields. Note the fields were declared as subfields of `summary`, so we call `addValue` on `summaryValue` instead of `target`.

```
@Entity
@Indexed
public class Invoice {

    @Id
    @GeneratedValue
    private Integer id;

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

    // Getters and setters
    // ...

}
```

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

Here is an example of what an indexed document would look like, with the Elasticsearch backend:

```
{
  "summary": {
    "total": 38.96,
    "books": 30.97,
    "shipping": 7.99
  }
}
```

12.3.2. Passing parameters

There are two ways to pass parameters to property bridges:

- One is (mostly) limited to string parameters, but is trivial to implement.
- The other can allow any type of parameters, but requires you to declare your own annotations.

Simple, string parameters

You can pass string parameters to the `@PropertyBinderRef` annotation and then use them later in the binder:

Example 12.13: Passing parameters to a `PropertyBinder` using the `@PropertyBinderRef` annotation

```
public class InvoiceLineItemsSummaryBinder implements PropertyBinder {

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

        String fieldName = context.params().get( "fieldName", String.class ); ①
        IndexSchemaObjectField summaryField = context.indexSchemaElement()
                .objectField( fieldName ); ②
```

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

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

    @SuppressWarnings("rawtypes")
    private static class Bridge implements PropertyBridge<List> {

        /* ... same implementation as before ... */

    }
}
```

① Use the binding context to get the parameter value.

The `param` method will throw an exception if the parameter has not been defined. Alternatively, use `paramOptional` to get an `java.util.Optional` that will be empty if the parameter has not been defined.

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

```
@Entity
@Indexed
public class Invoice {

    @Id
    @GeneratedValue
    private Integer id;

    @ElementCollection
    @OrderColumn
    @PropertyBinding(binder = @PropertyBinderRef( ①
            type = InvoiceLineItemsSummaryBinder.class,
            params = @Param(name = "fieldName", value = "itemSummary")))
    private List<InvoiceLineItem> lineItems = new ArrayList<>();

    // Getters and setters
    // ...

}
```

① Define the binder to use on the property, setting the `fieldName` parameter.

Parameters with custom annotations

You can pass parameters of any type to the bridge by defining a custom annotation with attributes:

Example 12.14: Passing parameters to a `PropertyBinder` using a custom annotation

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

    String fieldName() default ""; ⑤

    class Processor ⑥
            implements PropertyMappingAnnotationProcessor<InvoiceLineItemsSummaryBinding> { ⑦
        @Override
```

```
        public void process(PropertyMappingStep mapping,
                InvoiceLineItemsSummaryBinding annotation,
                PropertyMappingAnnotationProcessorContext context) {
            InvoiceLineItemsSummaryBinder binder = new InvoiceLineItemsSummaryBinder(); ⑧
            if ( !annotation.fieldName().isEmpty() ) { ⑨
                binder.fieldName( annotation.fieldName() );
            }
            mapping.binder( binder ); ⑩
        }
    }
}
```

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

② Since we're defining a property bridge, allow the annotation to target either methods (getters) or fields.

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

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

⑤ Define an attribute of type `String` to specify the field name.

⑥ Here the processor class is nested in the annotation class, because it is more convenient, but you are obviously free to implement it in a separate Java file.

⑦ The processor must implement the `PropertyMappingAnnotationProcessor` interface, setting its generic type argument to the type of the corresponding annotation.

⑧ In the annotation processor, instantiate the binder.

⑨ Process the annotation attributes and pass the data to the binder.

Here we're using a setter, but passing the data through the constructor would work, too.

⑩ Apply the binder to the property.

```
public class InvoiceLineItemsSummaryBinder implements PropertyBinder {

    private String fieldName = "summary";

    public InvoiceLineItemsSummaryBinder fieldName(String fieldName) { ①
        this.fieldName = fieldName;
        return this;
    }

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

        IndexSchemaObjectField summaryField = context.indexSchemaElement()
                .objectField( this.fieldName ); ②

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

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

    @SuppressWarnings("rawtypes")
    private static class Bridge implements PropertyBridge<List> {

        /* ... same implementation as before ... */

    }
}
```

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

② In the `bind` method, use the value of parameters. Here use the `fieldName` parameter to set the field name, but we could pass

parameters for any purpose: defining the field as sortable, defining a normalizer, …

```
@Entity
@Indexed
public class Invoice {

    @Id
    @GeneratedValue
    private Integer id;

    @ElementCollection
    @OrderColumn
    @InvoiceLineItemsSummaryBinding( ①
            fieldName = "itemSummary"
    )
    private List<InvoiceLineItem> lineItems = new ArrayList<>();

    // Getters and setters
    // ...

}
```

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

12.3.3. Accessing the ORM session from the bridge

This feature is only available with the Hibernate ORM integration.

It **cannot** be used with the Standalone POJO Mapper in particular.

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

Example 12.15: Retrieving the ORM session from a `PropertyBridge`

```
private static class Bridge implements PropertyBridge<Object> {

    private final IndexFieldReference<String> field;

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

    @Override
    public void write(DocumentElement target, Object bridgedElement, PropertyBridgeWriteContext context) {
        Session session = context.extension( HibernateOrmExtension.get() ) ①
                .session(); ②
        // ... do something with the session ...
    }
}
```

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

12.3.4. Injecting beans into the binder

With compatible frameworks, Hibernate Search supports injecting beans into:

- the `PropertyMappingAnnotationProcessor` if you use custom annotations.
- the `PropertyBinder` if you use the `@PropertyBinding` annotation.

This only applies to beans instantiated through Hibernate Search's bean resolution. As a rule of thumb, if you need to call `new MyBinder()` explicitly at some point, the binder won't get auto-magically injected.

The context passed to the property binder's `bind` method also exposes a `beanResolver()` method to access the bean resolver and

instantiate beans explicitly.

See Bean injection for more details.

12.3.5. Programmatic mapping

You can apply a property bridge through the programmatic mapping too. Just pass an instance of the binder. You can pass arguments either through the binder's constructor, or through setters.

Example 12.16: Applying an `PropertyBinder` *with* `.binder(…)`

```
TypeMappingStep invoiceMapping = mapping.type( Invoice.class );
invoiceMapping.indexed();
invoiceMapping.property( "lineItems" )
        .binder( new InvoiceLineItemsSummaryBinder().fieldName( "itemSummary" ) );
```

12.3.6. Incubating features

> Features detailed below are *incubating*: they are still under active development.
>
> The usual compatibility policy does not apply: the contract of incubating elements (e.g. types, methods, configuration properties, etc.) may be altered in a backward-incompatible way — or even removed — in subsequent releases.
>
> You are encouraged to use incubating features so the development team can get feedback and improve them, but you should be prepared to update code which relies on them as needed.

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

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

- Getting the name of the property.
- Checking the type of the property.
- Getting accessors to properties.
- Detecting properties with markers. Markers are applied by specific annotations carrying a `@MarkerBinding` meta-annotation.

See the javadoc for more information.

Below is an example of the simplest use of this metadata, getting the property name and using it as a field name.

Example 12.17: Naming a field after the property being bound in a `PropertyBinder`

```
public class InvoiceLineItemsSummaryBinder implements PropertyBinder {

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

        PojoModelProperty bridgedElement = context.bridgedElement(); ①
        IndexSchemaObjectField summaryField = context.indexSchemaElement()
                .objectField( bridgedElement.name() ); ②

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

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

```
        @SuppressWarnings("rawtypes")
        private static class Bridge implements PropertyBridge<List> {

            /* ... same implementation as before ... */

        }
    }
```

① Use the binding context to get the bridged element.
② Use the name of the property as the name of a newly declared index field.

```
    @Entity
    @Indexed
    public class Invoice {

        @Id
        @GeneratedValue
        private Integer id;

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

        // Getters and setters
        // ...

    }
```

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

Here is an example of what an indexed document would look like, with the Elasticsearch backend:

```
    {
      "lineItems": {
        "total": 38.96,
        "books": 30.97,
        "shipping": 7.99
      }
    }
```

12.4. Type bridge

12.4.1. Basics

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

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

Implementing a type bridge requires two components:

1. A custom implementation of `TypeBinder`, to bind the bridge to a type at bootstrap. This involves declaring the properties of the type that will be used, declaring the index fields that will be populated along with their type, and instantiating the type bridge.
2. A custom implementation of `TypeBridge`, to perform the conversion at runtime. This involves extracting data from an instance of the type, transforming the data if necessary, and pushing it to index fields.

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

Example 12.18: Implementing and using a `TypeBridge`

```
    public class FullNameBinder implements TypeBinder { ①
```

```
        @Override
        public void bind(TypeBindingContext context) { ②
            context.dependencies() ③
                    .use( "firstName" )
                    .use( "lastName" );

            IndexFieldReference<String> fullNameField = context.indexSchemaElement() ④
                    .field( "fullName", f -> f.asString().analyzer( "name" ) ) ⑤
                    .toReference();

            context.bridge( ⑥
                    Author.class, ⑦
                    new Bridge( ⑧
                            fullNameField ⑨
                    )
            );
        }

        // ... class continues below
```

① The binder must implement the `TypeBinder` interface.

② Implement the `bind` method in the binder.

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

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

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

⑥ Call `context.bridge(…)` to define the type bridge to use.

⑦ Pass the expected type of the entity.

⑧ Pass the type bridge instance.

⑨ Pass a reference to the `fullName` field to the bridge.

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

        private static class Bridge ①
                implements TypeBridge<Author> { ②

            private final IndexFieldReference<String> fullNameField;

            private Bridge(IndexFieldReference<String> fullNameField) { ③
                this.fullNameField = fullNameField;
            }

            @Override
            public void write(
                    DocumentElement target,
                    Author author,
                    TypeBridgeWriteContext context) { ④
                String fullName = author.getLastName() + " " + author.getFirstName(); ⑤
                target.addValue( this.fullNameField, fullName ); ⑥
            }
        }
    }
```

① Here the bridge class is nested in the binder class, because it is more convenient, but you are obviously free to implement it as you wish: as a lambda expression, in a separate Java file…

② The bridge must implement the `TypeBridge` interface. One generic type argument must be provided: the type of the "bridged element".

③ The bridge stores references to the fields — it will need them when indexing.

④ Implement the `write` method in the bridge. This method is called on indexing.

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

⑥ Set the value of the `fullName` field. Note the `fullName` field was declared at the root, so we call `addValue` directly on the `target` argument.

```
@Entity
@Indexed
@TypeBinding(binder = @TypeBinderRef(type = FullNameBinder.class)) ①
public class Author {

    @Id
    @GeneratedValue
    private Integer id;

    private String firstName;

    private String lastName;

    @GenericField ②
    private LocalDate birthDate;

    // Getters and setters
    // ...

}
```

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

② It is still possible to map properties directly using other annotations, as long as index field names are distinct from the names used in the type binder. But no annotation is necessary on the `firstName` and `lastName` properties: these are already handled by the bridge.

Here is an example of what an indexed document would look like, with the Elasticsearch backend:

```
{
  "fullName": "Asimov Isaac"
}
```

12.4.2. Passing parameters

There are two ways to pass parameters to type bridges:

- One is (mostly) limited to string parameters, but is trivial to implement.
- The other can allow any type of parameters, but requires you to declare your own annotations.

Simple, string parameters

You can pass string parameters to the `@TypeBinderRef` annotation and then use them later in the binder:

Example 12.19: Passing parameters to a `TypeBinder` using the `@TypeBinderRef` annotation

```
public class FullNameBinder implements TypeBinder {

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

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

        IndexFieldReference<String> fullNameSortField = null;
        String sortField = context.params().get( "sortField", String.class ); ①
        if ( "true".equalsIgnoreCase( sortField ) ) { ②
            fullNameSortField = context.indexSchemaElement()
                    .field(
                            "fullName_sort",
                            f -> f.asString().normalizer( "name" ).sortable( Sortable.YES )
                    )
                    .toReference();
```

```
        }

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

    private static class Bridge implements TypeBridge<Author> {

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

        private Bridge(IndexFieldReference<String> fullNameField,
                IndexFieldReference<String> fullNameSortField) { ②
            this.fullNameField = fullNameField;
            this.fullNameSortField = fullNameSortField;
        }

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

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

① Use the binding context to get the parameter value.

The `param` method will throw an exception if the parameter has not been defined. Alternatively, use `paramOptional` to get an `java.util.Optional` that will be empty if the parameter has not been defined.

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

```
@Entity
@Indexed
@TypeBinding(binder = @TypeBinderRef(type = FullNameBinder.class, ①
        params = @Param(name = "sortField", value = "true")))
public class Author {

    @Id
    @GeneratedValue
    private Integer id;

    private String firstName;

    private String lastName;

    // Getters and setters
    // ...

}
```

① Define the binder to use on the type, setting the `sortField` parameter.

Parameters with custom annotations

You can pass parameters of any type to the bridge by defining a custom annotation with attributes:

Example 12.20: Passing parameters to a `TypeBinder` using a custom annotation

```
@Retention(RetentionPolicy.RUNTIME) ①
@Target({ ElementType.TYPE }) ②
@TypeMapping(processor = @TypeMappingAnnotationProcessorRef(type = FullNameBinding.Processor.class)) ③
@Documented ④
public @interface FullNameBinding {

    boolean sortField() default false; ⑤

    class Processor ⑥
            implements TypeMappingAnnotationProcessor<FullNameBinding> { ⑦
        @Override
        public void process(TypeMappingStep mapping, FullNameBinding annotation,
                TypeMappingAnnotationProcessorContext context) {
            FullNameBinder binder = new FullNameBinder() ⑧
                    .sortField( annotation.sortField() ); ⑨
            mapping.binder( binder ); ⑩
        }
    }
}
```

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

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

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

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

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

⑥ Here the processor class is nested in the annotation class, because it is more convenient, but you are obviously free to implement it in a separate Java file.

⑦ The processor must implement the `TypeMappingAnnotationProcessor` interface, setting its generic type argument to the type of the corresponding annotation.

⑧ In the annotation processor, instantiate the binder.

⑨ Process the annotation attributes and pass the data to the binder.
Here we're using a setter, but passing the data through the constructor would work, too.

⑩ Apply the binder to the type.

```
public class FullNameBinder implements TypeBinder {

    private boolean sortField;

    public FullNameBinder sortField(boolean sortField) { ①
        this.sortField = sortField;
        return this;
    }

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

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

        IndexFieldReference<String> fullNameSortField = null;
        if ( this.sortField ) { ②
            fullNameSortField = context.indexSchemaElement()
                    .field(
                            "fullName_sort",
                            f -> f.asString().normalizer( "name" ).sortable( Sortable.YES )
                    )
                    .toReference();
        }
```

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

        private static class Bridge implements TypeBridge<Author> {

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

            private Bridge(IndexFieldReference<String> fullNameField,
                    IndexFieldReference<String> fullNameSortField) { ②
                this.fullNameField = fullNameField;
                this.fullNameSortField = fullNameSortField;
            }

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

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

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

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

```
    @Entity
    @Indexed
    @FullNameBinding(sortField = true) ①
    public class Author {

        @Id
        @GeneratedValue
        private Integer id;

        private String firstName;

        private String lastName;

        // Getters and setters
        // ...

    }
```

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

12.4.3. Accessing the ORM session from the bridge

This feature is only available with the Hibernate ORM integration.

It **cannot** be used with the Standalone POJO Mapper in particular.

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

Example 12.21: Retrieving the ORM session from a `TypeBridge`

```
private static class Bridge implements TypeBridge<Object> {

    private final IndexFieldReference<String> field;

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

    @Override
    public void write(DocumentElement target, Object bridgedElement, TypeBridgeWriteContext context) {
        Session session = context.extension( HibernateOrmExtension.get() ) ①
                .session(); ②
        // ... do something with the session ...
    }
}
```

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

② Retrieve the `Session` from the extended context.

12.4.4. Injecting beans into the binder

With compatible frameworks, Hibernate Search supports injecting beans into:

- the `TypeMappingAnnotationProcessor` if you use custom annotations.
- the `TypeBinder` if you use the `@TypeBinding` annotation.

This only applies to beans instantiated through Hibernate Search's bean resolution. As a rule of thumb, if you need to call `new MyBinder()` explicitly at some point, the binder won't get auto-magically injected.

The context passed to the routing key binder's `bind` method also exposes a `beanResolver()` method to access the bean resolver and instantiate beans explicitly.

See Bean injection for more details.

12.4.5. Programmatic mapping

You can apply a type bridge through the programmatic mapping too. Just pass an instance of the binder. You can pass arguments either through the binder's constructor, or through setters.

Example 12.22: Applying a `TypeBinder` with `.binder(…)`

```
TypeMappingStep authorMapping = mapping.type( Author.class );
authorMapping.indexed();
authorMapping.binder( new FullNameBinder().sortField( true ) );
```

12.4.6. Incubating features

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

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

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

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

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

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

See the javadoc for more information.

12.5. Identifier bridge

12.5.1. Basics

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

Implementing an identifier bridge boils down to implementing two methods:

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

Below is an example of a custom identifier bridge that converts a custom `BookId` type to its string representation and back:

Example 12.23: Implementing and using an `IdentifierBridge`

```
public class BookIdBridge implements IdentifierBridge<BookId> { ①

    @Override
    public String toDocumentIdentifier(BookId value,
            IdentifierBridgeToDocumentIdentifierContext context) { ②
        return value.getPublisherId() + "/" + value.getPublisherSpecificBookId();
    }

    @Override
    public BookId fromDocumentIdentifier(String documentIdentifier,
            IdentifierBridgeFromDocumentIdentifierContext context) { ③
        String[] split = documentIdentifier.split( "/" );
        return new BookId( Long.parseLong( split[0] ), Long.parseLong( split[1] ) );
    }

}
```

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

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

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

```
@Entity
@Indexed
public class Book {

    @EmbeddedId
    @DocumentId( ①
            identifierBridge = @IdentifierBridgeRef(type = BookIdBridge.class) ②
    )
    private BookId id = new BookId();

    private String title;

    // Getters and setters
    // ...

}
```

① Map the property to the document identifier.

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

12.5.2. Type resolution

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

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

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

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

12.5.3. Compatibility across indexes with `isCompatibleWith()`

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

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

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

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

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

Example 12.24: Implementing `isCompatibleWith` to support multi-index search

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

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

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

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

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

12.5.4. Parsing identifier's string representation with `parseIdentifierLiteral(..)`

In some scenarios, Hibernate Search may need to parse a string representation of an identifier, e.g. when the `ValueModel.STRING` is used in the matching clause of an identifier match predicate.

With a custom identifier bridge, Hibernate Search cannot automatically parse such identifier literals by default. To address this, `parseIdentifierLiteral(..)` can be implemented.

Example 12.25: Implementing `parseIdentifierLiteral(..)`

```
public class BookIdBridge implements IdentifierBridge<BookId> { ①
```

```
        // Implement mandatory toDocumentIdentifier/fromDocumentIdentifier ...
        // ...

        @Override
        public BookId parseIdentifierLiteral(String value) { ②
            if ( value == null ) {
                return null;
            }
            String[] parts = value.split( "/" );
            if ( parts.length != 2 ) {
                throw new IllegalArgumentException( "BookId string literal must be in a `pubId/bookId` format." );
            }
            return new BookId( Long.parseLong( parts[0] ), Long.parseLong( parts[1] ) );
        }
    }
```

① Start implementing the identifier bridge as usual.

② Implement `parseIdentifierLiteral(..)` to convert a string value to a `BookId`.

```
    List<Book> result = searchSession.search( Book.class )
            .where( f -> f.id().matching( "1/42", ValueModel.STRING ) ) ①
            .fetchHits( 20 );
```

① Use the `ValueModel.STRING` and a string representation of the identifier in the identifier match predicate.

12.5.5. Configuring the bridge more finely with `IdentifierBinder`

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

Example 12.26: Implementing an `IdentifierBinder`

```
    public class BookIdBinder implements IdentifierBinder { ①

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

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

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

① The binder must implement the `IdentifierBinder` interface.

② Implement the `bind` method.

③ Call `context.bridge(…)` to define the identifier bridge to use.

④ Pass the expected type of property values.

⑤ Pass the identifier bridge instance.

⑥ The identifier bridge must still be implemented.
Here the bridge class is nested in the binder class, because it is more convenient, but you are obviously free to implement it as

you wish: as a lambda expression, in a separate Java file…

```
@Entity
@Indexed
public class Book {

    @EmbeddedId
    @DocumentId( ①
            identifierBinder = @IdentifierBinderRef(type = BookIdBinder.class) ②
    )
    private BookId id = new BookId();

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

    // Getters and setters
    // ...

}
```

① Map the property to the document identifier.

② Instruct Hibernate Search to use our custom identifier binder. Note the use of `identifierBinder` instead of `identifierBridge`. It is also possible to reference the binder by its name, in the case of a CDI/Spring bean.

12.5.6. Passing parameters

There are two ways to pass parameters to identifier bridges:

- One is (mostly) limited to string parameters, but is trivial to implement.
- The other can allow any type of parameters, but requires you to declare your own annotations.

Simple, string parameters

You can pass string parameters to the `@IdentifierBinderRef` annotation and then use them later in the binder:

Example 12.27: Passing parameters to an `IdentifierBridge` using the `@IdentifierBinderRef` annotation

```
public class OffsetIdentifierBridge implements IdentifierBridge<Integer> { ①

    private final int offset;

    public OffsetIdentifierBridge(int offset) { ②
        this.offset = offset;
    }

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

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

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

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

```
public class OffsetIdentifierBinder implements IdentifierBinder {

    @Override
    public void bind(IdentifierBindingContext<?> context) {
```

```
        String offset = context.params().get( "offset", String.class ); ①
        context.bridge(
                Integer.class,
                new OffsetIdentifierBridge( Integer.parseInt( offset ) ) ②
        );
    }
}
```

① Use the binding context to get the parameter value.

The `param` method will throw an exception if the parameter has not been defined. Alternatively, use `paramOptional` to get an `java.util.Optional` that will be empty if the parameter has not been defined.

② Pass the parameter value as an argument to the bridge constructor.

```
@Entity
@Indexed
public class Book {

    @Id
    // DB identifiers start at 0, but index identifiers start at 1
    @DocumentId(identifierBinder = @IdentifierBinderRef( ①
            type = OffsetIdentifierBinder.class,
            params = @Param(name = "offset", value = "1")))
    private Integer id;

    private String title;

    // Getters and setters
    // ...

}
```

① Define the binder to use on the identifier, setting the parameter.

Parameters with custom annotations

You can pass parameters of any type to the bridge by defining a custom annotation with attributes:

Example 12.28: Passing parameters to an `IdentifierBridge` using a custom annotation

```
public class OffsetIdentifierBridge implements IdentifierBridge<Integer> { ①

    private final int offset;

    public OffsetIdentifierBridge(int offset) { ②
        this.offset = offset;
    }

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

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

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

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

```
@Retention(RetentionPolicy.RUNTIME) ①
@Target({ ElementType.METHOD, ElementType.FIELD }) ②
@PropertyMapping(processor = @PropertyMappingAnnotationProcessorRef( ③
```

```
        type = OffsetDocumentId.Processor.class
))
@Documented ④
public @interface OffsetDocumentId {

    int offset(); ⑤

    class Processor ⑥
            implements PropertyMappingAnnotationProcessor<OffsetDocumentId> { ⑦
        @Override
        public void process(PropertyMappingStep mapping, OffsetDocumentId annotation,
                PropertyMappingAnnotationProcessorContext context) {
            OffsetIdentifierBridge bridge = new OffsetIdentifierBridge( ⑧
                    annotation.offset()
            );
            mapping.documentId() ⑨
                    .identifierBridge( bridge ); ⑩
        }
    }
}
```

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

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

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

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

⑤ Define custom attributes to configure the value bridge. Here we define an offset that the bridge should add to entity identifiers.

⑥ Here the processor class is nested in the annotation class, because it is more convenient, but you are obviously free to implement it in a separate Java file.

⑦ The processor must implement the `PropertyMappingAnnotationProcessor` interface, setting its generic type argument to the type of the corresponding annotation.

⑧ In the `process` method, instantiate the bridge and pass the annotation attribute as constructor argument.

⑨ Declare that this property is to be used to generate the document identifier.

⑩ Instruct Hibernate Search to use our bridge to convert between the property and the document identifiers. Alternatively, we could pass an identifier binder instead, using the `identifierBinder()` method.

```
@Entity
@Indexed
public class Book {

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

    private String title;

    // Getters and setters
    // ...

}
```

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

12.5.7. Accessing the ORM session or session factory from the bridge

This feature is only available with the Hibernate ORM integration.

It **cannot** be used with the Standalone POJO Mapper in particular.

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

Example 12.29: Retrieving the ORM session or session factory from an `IdentifierBridge`

```
public class MyDataIdentifierBridge implements IdentifierBridge<MyData> {

    @Override
    public String toDocumentIdentifier(MyData propertyValue, IdentifierBridgeToDocumentIdentifierContext
context) {
        SessionFactory sessionFactory = context.extension( HibernateOrmExtension.get() ) ①
                .sessionFactory(); ②
        // ... do something with the factory ...
    }

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

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

12.5.8. Injecting beans into the bridge or binder

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

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

The context passed to the identifier binder's `bind` method also exposes a `beanResolver()` method to access the bean resolver and instantiate beans explicitly.

See Bean injection for more details.

12.5.9. Programmatic mapping

You can apply an identifier bridge through the programmatic mapping too. Just pass an instance of the bridge.

Example 12.30: Applying an `IdentifierBridge` with `.identifierBridge(…)`

```
TypeMappingStep bookMapping = mapping.type( Book.class );
bookMapping.indexed();
bookMapping.property( "id" )
        .documentId().identifierBridge( new BookIdBridge() );
```

Similarly, you can pass a binder instance:

Example 12.31: Applying an `IdentifierBinder` with `.identifierBinder(…)`

```
TypeMappingStep bookMapping = mapping.type( Book.class );
bookMapping.indexed();
bookMapping.property( "id" )
        .documentId().identifierBinder( new BookIdBinder() );
```

12.5.10. Incubating features

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

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

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

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

See the javadoc for more information.

12.6. Routing bridge

12.6.1. Basics

A routing bridge is a pluggable component that defines, at runtime, whether an entity should be indexed and to which shard the corresponding indexed document should be routed. It is applied to an indexed entity type with the `@Indexed` annotation, using its `routingBinder` attribute (`@Indexed(routingBinder = …)`).

Implementing a routing bridge requires two components:

1. A custom implementation of `RoutingBinder`, to bind the bridge to an indexed entity type at bootstrap. This involves declaring the properties of the indexed entity type that will be used by the routing bridge and instantiating the routing bridge.
2. A custom implementation of `RoutingBridge`, to route entities to the index at runtime. This involves extracting data from an instance of the type, transforming the data if necessary, and defining the current route (or marking the entity as "not indexed").

   If routing can change during the lifetime of an entity instance, you will also need to define the potential previous routes, so that Hibernate Search can find and delete previous documents indexed for this entity instance.

In the sections below, you will find examples for the main use cases:

- Using a routing bridge for conditional indexing
- Using a routing bridge to control routing to index shards

12.6.2. Using a routing bridge for conditional indexing

Below is a first example of a custom routing bridge that disables indexing for instances of the `Book` class if their status is `ARCHIVED`.

Example 12.32: Implementing and using a `RoutingBridge` for conditional indexing

```
public class BookStatusRoutingBinder implements RoutingBinder { ①

    @Override
    public void bind(RoutingBindingContext context) { ②
        context.dependencies() ③
                .use( "status" );

        context.bridge( ④
                Book.class, ⑤
                new Bridge() ⑥
        );
    }

    // ... class continues below
```

① The binder must implement the `RoutingBinder` interface.
② Implement the `bind` method in the binder.
③ Declare the dependencies of the bridge, i.e. the parts of the entity instances that the bridge will actually use. See Declaring dependencies to bridged elements for more information about declaring dependencies.
④ Call `context.bridge(…)` to define the routing bridge to use.
⑤ Pass the expected type of indexed entities.
⑥ Pass the routing bridge instance.

```
        // ... class BookStatusRoutingBinder (continued)

        public static class Bridge ①
                implements RoutingBridge<Book> { ②
            @Override
            public void route(DocumentRoutes routes, Object entityIdentifier, ③
                    Book indexedEntity, RoutingBridgeRouteContext context) {
                switch ( indexedEntity.getStatus() ) { ④
                    case PUBLISHED:
                        routes.addRoute(); ⑤
                        break;
                    case ARCHIVED:
                        routes.notIndexed(); ⑥
                        break;
                }
            }

            @Override
            public void previousRoutes(DocumentRoutes routes, Object entityIdentifier, ⑦
                    Book indexedEntity, RoutingBridgeRouteContext context) {
                routes.addRoute(); ⑧
            }
        }
    }
```

① Here the bridge class is nested in the binder class, because it is more convenient, but you are obviously free to implement it as you wish: as a lambda expression, in a separate Java file…

② The bridge must implement the `RoutingBridge` interface.

③ Implement the `route(…)` method in the bridge. This method is called on indexing.

④ Extract data from the bridged element and inspect it.

⑤ If the `Book` status is `PUBLISHED`, then we want to proceed with indexing: add a route so that Hibernate Search indexes the entity as usual.

⑥ If the `Book` status is `ARCHIVED`, then we don't want to index it: call `notIndexed()` so that Hibernate Search knows it should **not** index the entity.

⑦ When a book gets archived, there might be a previously indexed document that needs to be deleted. The `previousRoutes(…)` method allows you to tell Hibernate Search where this document can possibly be. When necessary, Hibernate Search will follow each given route, look for documents corresponding to this entity, and delete them.

⑧ In this case, routing is very simple: there is only one possible previous route, so we only register that route.

```
    @Entity
    @Indexed(routingBinder = @RoutingBinderRef(type = BookStatusRoutingBinder.class)) ①
    public class Book {

        @Id
        private Integer id;

        private String title;

        @Basic(optional = false)
        @KeywordField ②
        private Status status;

        // Getters and setters
        // ...

    }
```

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

② Properties used in the bridge can still be mapped as index fields, but they don't have to be.

12.6.3. Using a routing bridge to control routing to index shards

For a preliminary introduction to sharding, including how it works in Hibernate Search and what its limitations are, see Sharding and routing.

Routing bridges can also be used to control routing to index shards.

Below is an example of a custom routing bridge that uses the `genre` property of the `Book` class as a routing key. See Routing for an example of how to use routing in search queries, with the same mapping as the example below.

Example 12.33: Implementing and using a `RoutingBridge` to control routing to index shards

```
public class BookGenreRoutingBinder implements RoutingBinder { ①

    @Override
    public void bind(RoutingBindingContext context) { ②
        context.dependencies() ③
                .use( "genre" );

        context.bridge( ④
                Book.class, ⑤
                new Bridge() ⑥
        );
    }

    // ... class continues below
```

① The binder must implement the `RoutingBinder` interface.

② Implement the `bind` method in the binder.

③ Declare the dependencies of the bridge, i.e. the parts of the entity instances that the bridge will actually use. See Declaring dependencies to bridged elements for more information about declaring dependencies.

④ Call `context.bridge(…)` to define the routing bridge to use.

⑤ Pass the expected type of indexed entities.

⑥ Pass the routing bridge instance.

```
    // ... class BookGenreRoutingBinder (continued)

    public static class Bridge implements RoutingBridge<Book> { ①
        @Override
        public void route(DocumentRoutes routes, Object entityIdentifier, ②
                Book indexedEntity, RoutingBridgeRouteContext context) {
            String routingKey = indexedEntity.getGenre().name(); ③
            routes.addRoute().routingKey( routingKey ); ④
        }

        @Override
        public void previousRoutes(DocumentRoutes routes, Object entityIdentifier, ⑤
                Book indexedEntity, RoutingBridgeRouteContext context) {
            for ( Genre possiblePreviousGenre : Genre.values() ) {
                String routingKey = possiblePreviousGenre.name();
                routes.addRoute().routingKey( routingKey ); ⑥
            }
        }
    }
}
```

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

② Implement the `route(…)` method in the bridge. This method is called on indexing.

③ Extract data from the bridged element and derive a routing key.

④ Add a route with the generated routing key. Hibernate Search will follow this route when adding/updating/deleting the entity in the index.

⑤ When the genre of a book changes, the route will change, and there it might be a previously indexed document in the index that needs to be deleted. The `previousRoutes(…)` method allows you to tell Hibernate Search where this document can possibly be. When necessary, Hibernate Search will follow each given route, look for documents corresponding to this entity, and delete them.

⑥ In this case, we simply don't know what the previous genre of the book was, so we tell Hibernate Search to follow all possible routes, one for every possible genre.

```
@Entity
@Indexed(routingBinder = @RoutingBinderRef(type = BookGenreRoutingBinder.class)) ①
public class Book {
```

```
    @Id
    private Integer id;

    private String title;

    @Basic(optional = false)
    @KeywordField ②
    private Genre genre;

    // Getters and setters
    // ...

}
```

① Apply the bridge using the `@Indexed` annotation.
② Properties used in the bridge can still be mapped as index fields, but they don't have to be.

Optimizing `previousRoutes(…)`

In some cases you might have more information than in the example above about the previous routes, and you can take advantage of that information to trigger fewer deletions in the index:

- If the routing key is derived from an immutable property, then you can be sure the route never changes. In that case, just call `route(…)` with the arguments passed to `previousRoutes(…)` to tell Hibernate Search that the previous route is the same as the current route, and Hibernate Search will skip the deletion.
- If the routing key is derived from a property that changes in a predictable way, e.g. a status that **always** goes from `DRAFT` to `PUBLISHED` to `ARCHIVED` and never goes back, then you can be sure the previous routes are those corresponding to the possible previous values. In that case, just add one route for each possible previous status, e.g. if the current status is `PUBLISHED` you only need to add a route for `DRAFT` and `PUBLISHED`, but not for `ARCHIVED`.

12.6.4. Passing parameters

There are two ways to pass parameters to routing bridges:

- One is (mostly) limited to string parameters, but is trivial to implement.
- The other can allow any type of parameters, but requires you to declare your own annotations.

Refer to this example for `TypeBinder`, which is fairly similar to what you'll need for a `RoutingBinder`.

12.6.5. Accessing the ORM session from the bridge

This feature is only available with the Hibernate ORM integration.

It **cannot** be used with the Standalone POJO Mapper in particular.

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

Example 12.34: Retrieving the ORM session from a `RoutingBridge`

```
private static class Bridge implements RoutingBridge<MyEntity> {

    @Override
    public void route(DocumentRoutes routes, Object entityIdentifier, MyEntity indexedEntity,
            RoutingBridgeRouteContext context) {
        Session session = context.extension( HibernateOrmExtension.get() ) ①
                .session(); ②
        // ... do something with the session ...
    }

    @Override
    public void previousRoutes(DocumentRoutes routes, Object entityIdentifier, MyEntity indexedEntity,
            RoutingBridgeRouteContext context) {
        // ...
    }
}
```

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

12.6.6. Injecting beans into the binder

With compatible frameworks, Hibernate Search supports injecting beans into:

- the `TypeMappingAnnotationProcessor` if you use custom annotations.
- the `RoutingBinder` if you use `@Indexed(routingBinder = …)`.

This only applies to beans instantiated through Hibernate Search's bean resolution. As a rule of thumb, if you need to call `new MyBinder()` explicitly at some point, the binder won't get auto-magically injected.

The context passed to the routing binder's `bind` method also exposes a `beanResolver()` method to access the bean resolver and instantiate beans explicitly.

See Bean injection for more details.

12.6.7. Programmatic mapping

You can apply a routing key bridge through the programmatic mapping too. Just pass an instance of the binder.

Example 12.35: Applying an `RoutingBinder` with `.binder(…)`

```
TypeMappingStep bookMapping = mapping.type( Book.class );
bookMapping.indexed()
        .routingBinder( new BookStatusRoutingBinder() );
bookMapping.property( "status" ).keywordField();
```

12.6.8. Incubating features

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

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

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

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

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

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

See the javadoc for more information.

12.7. Declaring dependencies to bridged elements

12.7.1. Basics

In order to keep the index synchronized, Hibernate Search needs to be aware of all the entity properties that are used to produce indexed documents, so that it can trigger reindexing when they change.

When using a type bridge or a property bridge, the bridge itself decides which entity properties to access during indexing. Thus, it needs to let Hibernate Search know of its "dependencies" (the entity properties it may access).

This is done through a dedicated DSL, accessible from the `bind(…)` method of `TypeBinder` and `PropertyBinder`.

Below is an example of a type binder that expects to be applied to the `ScientificPaper` type, and declares a dependency to the paper author's last name and first name.

Example 12.36: Declaring dependencies in a bridge

```
public class AuthorFullNameBinder implements TypeBinder {

    @Override
    public void bind(TypeBindingContext context) {
        context.dependencies() ①
                .use( "author.firstName" ) ②
                .use( "author.lastName" ); ③

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

        context.bridge( Book.class, new Bridge( authorFullNameField ) );
    }

    private static class Bridge implements TypeBridge<Book> {

        // ...
    }
}
```

① Start the declaration of dependencies.
② Declare that the bridge will access the paper's `author` property, then the author's `firstName` property.
③ Declare that the bridge will access the paper's `author` property, then the author's `lastName` property.

The above should be enough to get started, but if you want to know more, here are a few facts about declaring dependencies.

Paths are relative to the bridged element

For example:

- for a type bridge on type `ScientificPaper`, path `author` will refer to the value of property `author` on `ScientificPaper` instances.
- for a property bridge on the property `author` of `ScientificPaper`, path `name` will refer to the value of property `name` on `Author` instances.

Every component of given paths will be considered as a dependency

You do not need to declare any parent path.

For example, if the path `myProperty.someOtherProperty` is declared as used, Hibernate Search will automatically assume that `myProperty` is also used.

Only mutable properties need to be declared

If a property never, ever changes after the entity is first persisted, then it will never trigger reindexing and Hibernate Search does not need to know about the dependency.

If your bridge only relies on immutable properties, see `useRootOnly()`: declaring no dependency at all.

Associations included in dependency paths need to have an inverse side

If you declare a dependency that crosses entity boundaries through an association, and that association has no inverse side in the other entity, an exception will be thrown.

For example, when you declare a dependency to path `author.lastName`, Hibernate Search infers that whenever the last name of an author changes, its books need to be re-indexed. Thus, when it detects an author's last name changed, Hibernate Search will need to retrieve the books to reindex them. That's why the `author` association in entity `ScientificPaper` needs to have an inverse side in entity `Author`, e.g. a `books` association.

See Tuning when to trigger reindexing for more information about these constraints and how to address non-trivial models.

12.7.2. Traversing non-default containers (map keys, ...)

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

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

You are encouraged to use incubating features so the development team can get feedback and improve them, but you

> should be prepared to update code which relies on them as needed.

When a path element refers to a property of a container type (`List`, `Map`, `Optional`, …), the path will be implicitly resolved to elements of that container. For example `someMap.otherObject` will resolve to the `otherObject` property of the *values* (not the keys) of `someMap`.

If the default resolution is not what you need, you can explicitly control how to traverse containers by passing `PojoModelPath` objects instead of just strings:

Example 12.37: Declaring dependencies in a bridge with explicit container extractors

```
@Entity
@Indexed
@TypeBinding(binder = @TypeBinderRef(type = BookEditionsForSaleTypeBinder.class)) ①
public class Book {

    @Id
    @GeneratedValue
    private Integer id;

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

    @ElementCollection
    @JoinTable(
            name = "book_editionbyprice",
            joinColumns = @JoinColumn(name = "book_id")
    )
    @MapKeyJoinColumn(name = "edition_id")
    @Column(name = "price")
    @OrderBy("edition_id asc")
    @AssociationInverseSide(
            extraction = @ContainerExtraction(BuiltinContainerExtractors.MAP_KEY),
            inversePath = @ObjectPath(@PropertyValue(propertyName = "book"))
    )
    private Map<BookEdition, BigDecimal> priceByEdition = new LinkedHashMap<>(); ②

    public Book() {
    }

    // Getters and setters
    // ...

}
```

① Apply a custom bridge to the `ScientificPaper` entity.
② This (rather complex) map is the one we'll access in the custom bridge.

```
public class BookEditionsForSaleTypeBinder implements TypeBinder {

    @Override
    public void bind(TypeBindingContext context) {
        context.dependencies()
                .use( PojoModelPath.builder() ①
                        .property( "priceByEdition" ) ②
                        .value( BuiltinContainerExtractors.MAP_KEY ) ③
                        .property( "label" ) ④
                        .toValuePath() ); ⑤

        IndexFieldReference<String> editionsForSaleField = context.indexSchemaElement()
                .field( "editionsForSale", f -> f.asString().analyzer( "english" ) )
                .multiValued()
                .toReference();

        context.bridge( Book.class, new Bridge( editionsForSaleField ) );
    }

    private static class Bridge implements TypeBridge<Book> {

        private final IndexFieldReference<String> editionsForSaleField;
```

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

        @Override
        public void write(DocumentElement target, Book book, TypeBridgeWriteContext context) {
            for ( BookEdition edition : book.getPriceByEdition().keySet() ) { ⑥
                target.addValue( editionsForSaleField, edition.getLabel() );
            }
        }
    }
}
```

① Start building a `PojoModelPath`.

② Append the `priceByEdition` property (a `Map`) to the path.

③ Explicitly mention that the bridge will access *keys* from the `priceByEdition` map — the paper editions. Without this, Hibernate Search would have assumed that *values* are accessed.

④ Append the `label` property to the path. This is the `label` property in paper editions.

⑤ Create the path and pass it to `.use(…)` to declare the dependency.

⑥ This is the actual code that accesses the paths as declared above.

For property binders applied to a container property, you can control how to traverse the property itself by passing a container extractor path as the first argument to `use(…)`:

Example 12.38: Declaring dependencies in a bridge with explicit container extractors for the bridged property

```
@Entity
@Indexed
public class Book {

    @Id
    @GeneratedValue
    private Integer id;

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

    @ElementCollection
    @JoinTable(
            name = "book_editionbyprice",
            joinColumns = @JoinColumn(name = "book_id")
    )
    @MapKeyJoinColumn(name = "edition_id")
    @Column(name = "price")
    @OrderBy("edition_id asc")
    @AssociationInverseSide(
            extraction = @ContainerExtraction(BuiltinContainerExtractors.MAP_KEY),
            inversePath = @ObjectPath(@PropertyValue(propertyName = "book"))
    )
    @PropertyBinding(binder = @PropertyBinderRef(type = BookEditionsForSalePropertyBinder.class)) ①
    private Map<BookEdition, BigDecimal> priceByEdition = new LinkedHashMap<>();

    public Book() {
    }

    // Getters and setters
    // ...

}
```

① Apply a custom bridge to the `pricesByEdition` property of the `ScientificPaper` entity.

```
public class BookEditionsForSalePropertyBinder implements PropertyBinder {

    @Override
    public void bind(PropertyBindingContext context) {
```

```
            context.dependencies()
                    .use( ContainerExtractorPath.explicitExtractor( BuiltinContainerExtractors.MAP_KEY ), ①
                            "label" ); ②

            IndexFieldReference<String> editionsForSaleField = context.indexSchemaElement()
                    .field( "editionsForSale", f -> f.asString().analyzer( "english" ) )
                    .multiValued()
                    .toReference();

            context.bridge( Map.class, new Bridge( editionsForSaleField ) );
        }

        @SuppressWarnings("rawtypes")
        private static class Bridge implements PropertyBridge<Map> {

            private final IndexFieldReference<String> editionsForSaleField;

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

            @Override
            public void write(DocumentElement target, Map bridgedElement, PropertyBridgeWriteContext context) {
                @SuppressWarnings("unchecked")
                Map<BookEdition, ?> priceByEdition = (Map<BookEdition, ?>) bridgedElement;

                for ( BookEdition edition : priceByEdition.keySet() ) { ③
                    target.addValue( editionsForSaleField, edition.getLabel() );
                }
            }
        }
    }
```

① Explicitly mention that the bridge will access *keys* from the `priceByEdition` property — the paper editions. Without this, Hibernate Search would have assumed that *values* are accessed.

② Declare a dependency to the `label` property in paper editions.

③ This is the actual code that accesses the paths as declared above.

12.7.3. `useRootOnly()`: declaring no dependency at all

If your bridge only accesses immutable properties, then it's safe to declare that its only dependency is to the root object.

To do so, call `.dependencies().useRootOnly()`.

Without this call, Hibernate Search will suspect an oversight and will throw an exception on startup.

12.7.4. `fromOtherEntity(…)`: declaring dependencies using the inverse path

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

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

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

It is not always possible to represent the dependency as a path from the bridged element to the values accessed by the bridge.

In particular, when the bridge relies on other components (queries, services) to retrieve another entity, there may not even be a path from the bridge element to that entity. In this case, if there is an *inverse* path from the other entity to the bridged element, and the bridged element is an entity, you can simply declare the dependency from the other entity, as shown below.

Example 12.39: Declaring dependencies in a bridge using the inverse path

```
    @Entity
    @Indexed
```

```
@TypeBinding(binder = @TypeBinderRef(type = ScientificPapersReferencedByBinder.class)) ①
public class ScientificPaper {

    @Id
    private Integer id;

    private String title;

    @ManyToMany
    private List<ScientificPaper> references = new ArrayList<>();

    public ScientificPaper() {
    }

    // Getters and setters
    // ...

}
```

① Apply a custom bridge to the `ScientificPaper` type.

```
public class ScientificPapersReferencedByBinder implements TypeBinder {

    @Override
    public void bind(TypeBindingContext context) {
        context.dependencies()
                .fromOtherEntity( ScientificPaper.class, "references" ) ①
                .use( "title" ); ②

        IndexFieldReference<String> papersReferencingThisOneField = context.indexSchemaElement()
                .field( "referencedBy", f -> f.asString().analyzer( "english" ) )
                .multiValued()
                .toReference();

        context.bridge( ScientificPaper.class, new Bridge( papersReferencingThisOneField ) );
    }

    private static class Bridge implements TypeBridge<ScientificPaper> {

        private final IndexFieldReference<String> referencedByField;

        private Bridge(IndexFieldReference<String> referencedByField) { ②
            this.referencedByField = referencedByField;
        }

        @Override
        public void write(DocumentElement target, ScientificPaper paper, TypeBridgeWriteContext context) {
            for ( String referencingPaperTitle : findReferencingPaperTitles( context, paper ) ) { ③
                target.addValue( referencedByField, referencingPaperTitle );
            }
        }

        private List<String> findReferencingPaperTitles(TypeBridgeWriteContext context, ScientificPaper paper) {
            Session session = context.extension( HibernateOrmExtension.get() ).session();
            Query<String> query = session.createQuery(
                    "select p.title from ScientificPaper p where :this member of p.references",
                    String.class );
            query.setParameter( "this", paper );
            return query.list();
        }
    }
}
```

① Declare that this bridge relies on other entities of type `ScientificPaper`, and that those other entities reference the indexed entity through their `references` property.

② Declare which parts of the other entities are actually used by the bridge.

③ The bridge retrieves the other entity through a query, but then uses exclusively the parts that were declared previously.

Currently, dependencies declared this way will be ignored when the "other entity" gets deleted.

See HSEARCH-3567 to track progress on solving this problem.

12.8. Declaring and writing to index fields

12.8.1. Basics

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

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

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

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

Example 12.40: Declaring a simple index field and writing to that field

```
public class ISBNBinder implements PropertyBinder {

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

        IndexSchemaElement schemaElement = context.indexSchemaElement(); ①

        IndexFieldReference<String> field =
                schemaElement.field( ②
                        "isbn", ③
                        f -> f.asString() ④
                                .normalizer( "isbn" )
                )
                        .toReference(); ⑤

        context.bridge( ⑥
                ISBN.class, ⑦
                new ISBNBridge( field ) ⑧
        );
    }
}
```

① Get the `IndexNode`, the entry point to the index field declaration DSL.
② Declare a field.
③ Pass the name of the field.
④ Declare the type of the field. This is done through a lambda taking advantage of another DSL. See Defining index field types for more information.
⑤ Get a reference to the declared field.
⑥ Call `context.bridge(…)` to define the bridge to use.
⑦ Pass the expected type of values.
⑧ Pass the bridge instance.

```
private static class ISBNBridge implements PropertyBridge<ISBN> {

    private final IndexFieldReference<String> fieldReference;

    private ISBNBridge(IndexFieldReference<String> fieldReference) {
        this.fieldReference = fieldReference;
    }

    @Override
    public void write(DocumentElement target, ISBN bridgedElement, PropertyBridgeWriteContext context) {
        String indexedValue = /* ... (extraction of data, not relevant) ... */
```

```
            target.addValue( this.fieldReference, indexedValue ); ①
        }
    }
```

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

12.8.2. Type objects

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

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

Example 12.41: Re-using an index field type in multiple field declarations

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

        IndexSchemaElement schemaElement = context.indexSchemaElement();

        IndexFieldType<String> nameType = context.typeFactory() ①
                .asString() ②
                .analyzer( "name" )
                .toIndexFieldType(); ③

        context.bridge( Author.class, new Bridge(
                schemaElement.field( "firstName", nameType ) ④
                        .toReference(),
                schemaElement.field( "lastName", nameType ) ④
                        .toReference(),
                schemaElement.field( "fullName", nameType ) ④
                        .toReference()
        ) );
    }
```

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

12.8.3. Multivalued fields

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

In order to add multiple values to a field, this field must be marked as multivalued during its declaration:

Example 12.42: Declaring a field as multivalued

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

        IndexSchemaElement schemaElement = context.indexSchemaElement();

        context.bridge( Author.class, new Bridge(
                schemaElement.field( "names", f -> f.asString().analyzer( "name" ) )
                        .multiValued() ①
                        .toReference()
        ) );
    }
```

① Declare the field as multivalued.

12.8.4. Object fields

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

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

Object fields are declared similarly to value fields, with an additional step to declare each subfield, as shown below.

Example 12.43: Declaring an object field

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

    IndexSchemaElement schemaElement = context.indexSchemaElement();

    IndexSchemaObjectField summaryField =
            schemaElement.objectField( "summary" ); ①

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

    context.bridge( List.class, new Bridge(
            summaryField.toReference(), ②
            summaryField.field( "total", amountFieldType ) ③
                    .toReference(),
            summaryField.field( "books", amountFieldType ) ③
                    .toReference(),
            summaryField.field( "shipping", amountFieldType ) ③
                    .toReference()
    ) );
}
```

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

The subfields of an object field can include object fields.

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

Object fields as well as their subfields are each assigned a reference, which will be used by the bridge to write to documents, as shown in the example below.

Example 12.44: Writing to an object field

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

    BigDecimal total = BigDecimal.ZERO;
    BigDecimal books = BigDecimal.ZERO;
    BigDecimal shipping = BigDecimal.ZERO;
    /* ... (computation of amounts, not relevant) ... */

    DocumentElement summary = target.addObject( this.summaryField ); ①
```

```
        summary.addValue( this.totalField, total ); ②
        summary.addValue( this.booksField, books ); ②
        summary.addValue( this.shippingField, shipping ); ②
    }
```

① Add an object to the `summary` object field for the current document, and get a reference to that object.
② Add a value to the subfields for the object we just added. Note we're calling `addValue` on the object we just added, not on `target`.

12.8.5. Object structure

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

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

Example 12.45: Declaring an object field as nested

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

        IndexSchemaElement schemaElement = context.indexSchemaElement();

        IndexSchemaObjectField lineItemsField =
                schemaElement.objectField( ①
                        "lineItems", ②
                        ObjectStructure.NESTED ③
                )
                        .multiValued(); ④

        context.bridge( List.class, new Bridge(
                lineItemsField.toReference(), ⑤
                lineItemsField.field( "category", f -> f.asString() ) ⑥
                        .toReference(),
                lineItemsField.field( "amount", f -> f.asBigDecimal().decimalScale( 2 ) ) ⑦
                        .toReference()
        ) );
    }
```

① Declare an object field with `objectField`.
② Define the name of the object field.
③ Define the structure of the object field, here `NESTED`.
④ Define the object field as multivalued.
⑤ Get a reference to the declared object field and pass it to the bridge for later use.
⑥ Create subfields, get references to these fields and pass them to the bridge for later use.

12.8.6. Dynamic fields with field templates

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

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

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

A field template is always declared in a binder: either in a type binder or in a property binder. As for static fields, the entry point to declaring a template is the `IndexNode` passed to the binder's `bind(…)` method. A call to the `fieldTemplate` method on the schema element will declare a field template.

Assuming a field template was declared during binding, the bridge can then add dynamic fields to the `DocumentElement` when indexing, by calling `addValue` and passing the field name (as a string) and the field value.

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

Example 12.46: Declaring a field template and writing to a dynamic field

```
public class UserMetadataBinder implements PropertyBinder {

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

        IndexSchemaElement schemaElement = context.indexSchemaElement();

        IndexSchemaObjectField userMetadataField =
                schemaElement.objectField( "userMetadata" ); ①

        userMetadataField.fieldTemplate( ②
                "userMetadataValueTemplate", ③
                f -> f.asString().analyzer( "english" ) ④
        ); ⑤

        context.bridge( Map.class, new UserMetadataBridge(
                userMetadataField.toReference() ⑥
        ) );
    }
}
```

① Declare an object field with `objectField`. It's better to always host your dynamic fields on a dedicated object field, to avoid conflicts with other templates.
② Declare a field template with `fieldTemplate`.
③ Pass the **template** name—this is not the field name, and is only used to uniquely identify the template.
④ Define the field type.
⑤ On contrary to static field declarations, field template declarations do not return a field reference, because you won't need it when writing to the document.
⑥ Get a reference to the declared object field and pass it to the bridge for later use.

```
@SuppressWarnings("rawtypes")
private static class UserMetadataBridge implements PropertyBridge<Map> {

    private final IndexObjectFieldReference userMetadataFieldReference;

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

    @Override
    public void write(DocumentElement target, Map bridgedElement, PropertyBridgeWriteContext context) {
        @SuppressWarnings("unchecked")
        Map<String, String> userMetadata = (Map<String, String>) bridgedElement;

        DocumentElement indexedUserMetadata = target.addObject( userMetadataFieldReference ); ①

        for ( Map.Entry<String, String> entry : userMetadata.entrySet() ) {
            String fieldName = entry.getKey();
            String fieldValue = entry.getValue();
            indexedUserMetadata.addValue( fieldName, fieldValue ); ②
        }
    }
}
```

① Add an object to the `userMetadata` object field for the current document, and get a reference to that object.
② Add one field per user metadata entry, with the field name and field value defined by the user. Note that field names should usually be validated before that point, in order to avoid exotic characters (whitespaces, dots, …).

Though rarely necessary, you can also declare templates for object fields using the `objectFieldTemplate` methods.

It is also possible to add multiple fields with different types to the same object. To that end, make sure that the format of a field can be inferred from the field name. You can then declare multiple templates and assign a path pattern to each template, as shown below.

Example 12.47: Declaring multiple field templates with different types

```
public class MultiTypeUserMetadataBinder implements PropertyBinder {

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

        IndexSchemaElement schemaElement = context.indexSchemaElement();

        IndexSchemaObjectField userMetadataField =
                schemaElement.objectField( "multiTypeUserMetadata" ); ①

        userMetadataField.fieldTemplate( ②
                "userMetadataValueTemplate_int", ③
                f -> f.asInteger().sortable( Sortable.YES ) ④
        )
                .matchingPathGlob( "*_int" ); ⑤

        userMetadataField.fieldTemplate( ⑥
                "userMetadataValueTemplate_default",
                f -> f.asString().analyzer( "english" )
        );

        context.bridge( Map.class, new Bridge( userMetadataField.toReference() ) );
    }
}
```

① Declare an object field with `objectField`.
② Declare a field template for integer with `fieldTemplate`.
③ Pass the **template** name.
④ Define the field type as integer, sortable.
⑤ Assign a path pattern to the template, so that only fields ending with `_int` will be considered as integers.
⑥ Declare another field template, so that fields are considered as english text if they do not match the previous template.

```
@SuppressWarnings("rawtypes")
private static class Bridge implements PropertyBridge<Map> {

    private final IndexObjectFieldReference userMetadataFieldReference;

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

    @Override
    public void write(DocumentElement target, Map bridgedElement, PropertyBridgeWriteContext context) {
        @SuppressWarnings("unchecked")
        Map<String, Object> userMetadata = (Map<String, Object>) bridgedElement;

        DocumentElement indexedUserMetadata = target.addObject( userMetadataFieldReference ); ①

        for ( Map.Entry<String, Object> entry : userMetadata.entrySet() ) {
            String fieldName = entry.getKey();
            Object fieldValue = entry.getValue();
            indexedUserMetadata.addValue( fieldName, fieldValue ); ②
        }
    }
}
```

① Add an object to the `userMetadata` object field for the current document, and get a reference to that object.
② Add one field per user metadata entry, with the field name and field value defined by the user. Note that field values should be validated before that point; in this case, adding a field named `foo_int` with a value of type `String` will lead to a `SearchException` when indexing.

Precedence of field templates

Hibernate Search tries to match templates in the order they are declared, so you should always declare the templates with the most specific path pattern first.

Templates declared on a given schema element can be matched in children of that element. For example, if you declare templates at the root of your entity (through a type bridge), these templates will be implicitly available in every single property bridge of that entity. In such cases, templates declared in property bridges will take precedence over those declared in the type bridge.

12.9. Defining index field types

12.9.1. Basics

A specificity of Lucene-based search engines (including Elasticsearch) is that field types are much more complex than just a data type ("string", "integer", …).

When declaring a field, you must not only declare the data type, but also various characteristics that will define how the data is stored exactly: is the field sortable, is it projectable, is it analyzed and if so with which analyzer, …

Because of this complexity, when field types must be defined in the various binders (`ValueBinder`, `PropertyBinder`, `TypeBinder`), they are defined using a dedicated DSL.

The entry point to this DSL is the `IndexFieldTypeFactory`. The type factory is generally accessible though the context object passed to the binders (`context.typeFactory()`). In the case of `PropertyBinder` and `TypeBinder`, the type factory can also be passed to the lambda expression passed to the `field` method to define the field type inline.

The type factory exposes various `as*()` methods, for example `asString` or `asLocalDate`. These are the first steps of the type definition DSL, where the data type is defined. They return other steps, from which options can be set, such as the analyzer. See below for an example.

Example 12.48: Defining a field type

```
IndexFieldType<String> type = context.typeFactory() ①
        .asString() ②
        .normalizer( "isbn" ) ③
        .sortable( Sortable.YES ) ③
        .toIndexFieldType(); ④
```

① Get the `IndexFieldTypeFactory` from the binding context.
② Define the data type.
③ Define options. Available options differ based on the field type: for example, `normalizer` is available for `String` fields, but not for `Double` fields.
④ Get the index field type.

In `ValueBinder`, the call to `toIndexFieldType()` is omitted: `context.bridge(…)` expects to be passed the last DSL step, not a fully built type.

`toIndexFieldType()` is also omitted in the lambda expressions passed to the `field` method of the field declaration DSL.

12.9.2. Available data types

All available data types have a dedicated `as*()` method in `IndexFieldTypeFactory`. For details, see the javadoc of `IndexFieldTypeFactory`, or the backend-specific documentation:

- available data types in the Lucene backend
- available data types in the Elasticsearch backend

12.9.3. Available type options

Most of the options available in the index field type DSL are identical to the options exposed by `@*Field` annotations. See Field annotation attributes for details about them.

Other options are explained in the following sections.

12.9.4. DSL converter

This section is not relevant for `ValueBinder`: Hibernate Search sets the DSL converter automatically for value bridges, creating a DSL converter that simply delegates to the value bridge.

The various search DSLs expose some methods that expect a field value: `matching()`, `between()`, `atMost()`, `missingValue().use()`, … By default, the expected type will be the same as the data type, i.e. `String` if you called `asString()`, `LocalDate` if you called `asLocalDate()`, etc.

This can be annoying when the bridge converts values from a different type when indexing. For example, if the bridge converts an enum to a string when indexing, you probably don't want to pass a string to search predicates, but rather the enum.

By setting a DSL converter on a field type, it is possible to change the expected type of values passed to the various DSL, See below for an example.

Example 12.49: Assigning a DSL converter to a field type

```
IndexFieldType<String> type = context.typeFactory()
        .asString() ①
        .normalizer( "isbn" )
        .sortable( Sortable.YES )
        .dslConverter( ②
                ISBN.class, ③
                (value, convertContext) -> value.getStringValue() ④
        )
        .toIndexFieldType();
```

① Define the data type as `String`.
② Define a DSL converter that converts from `ISBN` to `String`. This converter will be used transparently by the search DSLs.
③ Define the input type as `ISBN` by passing `ISBN.class` as the first parameter.
④ Define how to convert an `ISBN` to a `String` by passing a converter as the second parameter.

```
ISBN expectedISBN = /* ... */
List<Book> result = searchSession.search( Book.class )
        .where( f -> f.match().field( "isbn" )
                .matching( expectedISBN ) ) ①
        .fetchHits( 20 );
```

① Thanks to the DSL converter, predicates targeting fields using our type accept `ISBN` values by default.

DSL converters can be disabled in the various DSLs where necessary. See Type of arguments passed to the DSL.

12.9.5. Projection converter

This section is not relevant for `ValueBinder`: Hibernate Search sets the projection converter automatically for value bridges, creating a projection converter that simply delegates to the value bridge.

By default, the type of values returned by field projections or aggregations will be the same as the data type of the corresponding field, i.e. `String` if you called `asString()`, `LocalDate` if you called `asLocalDate()`, etc.

This can be annoying when the bridge converts values from a different type when indexing. For example, if the bridge converts an enum to a string when indexing, you probably don't want projections to return a string, but rather the enum.

By setting a projection converter on a field type, it is possible to change the type of values returned by field projections or aggregations. See below for an example.

Example 12.50: Assigning a projection converter to a field type

```
IndexFieldType<String> type = context.typeFactory()
        .asString() ①
        .projectable( Projectable.YES )
        .projectionConverter( ②
                ISBN.class, ③
                (value, convertContext) -> ISBN.parse( value ) ④
        )
```

```
            .toIndexFieldType();
```

① Define the data type as `String`.
② Define a projection converter that converts from `String` to `ISBN`. This converter will be used transparently by the search DSLs.
③ Define the converted type as `ISBN` by passing `ISBN.class` as the first parameter.
④ Define how to convert a `String` to an `ISBN` by passing a converter as the second parameter.

```
List<ISBN> result = searchSession.search( Book.class )
        .select( f -> f.field( "isbn", ISBN.class ) ) ①
        .where( f -> f.matchAll() )
        .fetchHits( 20 );
```

① Thanks to the projection converter, fields using our type are projected to an `ISBN` by default.

Projection converters can be disabled in the projection DSL where necessary. See Type of projected values.

12.9.6. Backend-specific types

Backends define extensions to this DSL to define backend-specific types.

See:

- Lucene index field type DSL extension
- Elasticsearch index field type DSL extension

12.10. Defining named predicates

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

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

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

When implementing a `PropertyBinder` or `TypeBinder`, it is possible to assign "named predicates" to index schema elements (either the index root or an object field).

These named predicates will then be usable through the Search DSL, referencing them by name and optionally passing parameters. The main point is that the implementation is hidden from callers: they do not need to understand how data is indexed in order to use a named predicate.

Below is a simple example using the DSL to declare an object field and assign a named predicate to that field, in a property binder.

Example 12.51: Declaring a named predicate

```
/**
 * A binder for Stock Keeping Unit (SKU) identifiers, i.e. Strings with a specific format.
 */
public class SkuIdentifierBinder implements PropertyBinder {

    @Override
    public void bind(PropertyBindingContext context) {
        context.dependencies().useRootOnly();

        IndexSchemaObjectField skuIdObjectField = context.indexSchemaElement()
                .objectField( context.bridgedElement().name() );

        IndexFieldType<String> skuIdPartType = context.typeFactory()
                .asString().normalizer( "lowercase" ).toIndexFieldType();

        context.bridge( String.class, new Bridge(
                skuIdObjectField.toReference(),
                skuIdObjectField.field( "departmentCode", skuIdPartType ).toReference(),
                skuIdObjectField.field( "collectionCode", skuIdPartType ).toReference(),
```

```
                skuIdObjectField.field( "itemCode", skuIdPartType ).toReference()
        ) );

        skuIdObjectField.namedPredicate( ①
                "skuIdMatch", ②
                new SkuIdentifierMatchPredicateDefinition() ③
        );

        skuIdObjectField.namedPredicate( ①
                "skuIdMatch2", ②
                new TypedPredicateDefinition<SkuIdentifierBinder>() {

                    @Override
                    public SearchPredicate create(TypedPredicateDefinitionContext<SkuIdentifierBinder> context)
{
                        return null;
                    }

                    @Override
                    public Class<SkuIdentifierBinder> scopeRootType() {
                        return SkuIdentifierBinder.class;
                    }
                } ③
        );

        skuIdObjectField.namedPredicate( ①
                "skuIdMatch3", ②
                new TypedPredicateDefinition<>() {

                    @Override
                    public SearchPredicate create(TypedPredicateDefinitionContext<Object> context) {
                        return null;
                    }

                    @Override
                    public Class<Object> scopeRootType() {
                        return Object.class;
                    }
                } ③
        );
    }

    // ... class continues below
```

① The binder defines a named predicate. Note this predicate is assigned to an object field.

② The predicate name will be used to refer to this predicate when calling the named predicate. Since the predicate is assigned to an object field, callers will have to prefix the predicate name with the path to that object field.

③ The predicate definition will define how to create the predicate when searching.

```
// ... class SkuIdentifierBinder (continued)

private static class Bridge implements PropertyBridge<String> { ①

    private final IndexObjectFieldReference skuIdObjectField;
    private final IndexFieldReference<String> departmentCodeField;
    private final IndexFieldReference<String> collectionCodeField;
    private final IndexFieldReference<String> itemCodeField;

    private Bridge(IndexObjectFieldReference skuIdObjectField,
            IndexFieldReference<String> departmentCodeField,
            IndexFieldReference<String> collectionCodeField,
            IndexFieldReference<String> itemCodeField) {
        this.skuIdObjectField = skuIdObjectField;
        this.departmentCodeField = departmentCodeField;
        this.collectionCodeField = collectionCodeField;
        this.itemCodeField = itemCodeField;
    }

    @Override
    public void write(DocumentElement target, String skuId, PropertyBridgeWriteContext context) {
```

```
            DocumentElement skuIdObject = target.addObject( this.skuIdObjectField );②

            // An SKU identifier is formatted this way: "<department code>.<collection code>.<item code>".
            String[] skuIdParts = skuId.split( "\\." );
            skuIdObject.addValue( this.departmentCodeField, skuIdParts[0] ); ③
            skuIdObject.addValue( this.collectionCodeField, skuIdParts[1] ); ③
            skuIdObject.addValue( this.itemCodeField, skuIdParts[2] ); ③
        }
    }

    // ... class continues below
```

① Here the bridge class is nested in the binder class, because it is more convenient, but you are obviously free to implement it as you wish: as a lambda expression, in a separate Java file…

② The bridge creates an object to hold the various components of the SKU identifier.

③ The bridge populates the various components of the SKU identifier.

```
        // ... class SkuIdentifierBinder (continued)

        private static class SkuIdentifierMatchPredicateDefinition implements PredicateDefinition { ①
            @Override
            public SearchPredicate create(PredicateDefinitionContext context) {
                SearchPredicateFactory f = context.predicate(); ②

                String pattern = context.params().get( "pattern", String.class ); ③

                return f.and().with( and -> { ④
                    // An SKU identifier pattern is formatted this way: "<department code>.<collection code>.<item
    code>".
                    // Each part supports * and ? wildcards.
                    String[] patternParts = pattern.split( "\\." );
                    if ( patternParts.length > 0 ) {
                        and.add( f.wildcard()
                                .field( "departmentCode" ) ⑤
                                .matching( patternParts[0] ) );
                    }
                    if ( patternParts.length > 1 ) {
                        and.add( f.wildcard()
                                .field( "collectionCode" )
                                .matching( patternParts[1] ) );
                    }
                    if ( patternParts.length > 2 ) {
                        and.add( f.wildcard()
                                .field( "itemCode" )
                                .matching( patternParts[2] ) );
                    }
                } ).toPredicate(); ⑥
            }
        }
    }
```

① The predicate definition must implement the `PredicateDefinition` interface.

Here the predicate definition class is nested in the binder class, because it is more convenient, but you are obviously free to implement it in a separate java file.

② The context passed to the definition exposes the predicate factory, which is the entry point to the predicate DSL, used to create predicates.

③ The definition can access parameters that are passed when calling the named predicates.

The `param` method will throw an exception if the parameter has not been defined. Alternatively, use `paramOptional` to get an `java.util.Optional` that will be empty if the parameter has not been defined.

④ The definition uses the predicate factory to create predicates. In this example, this implementation transforms a pattern with a custom format into three patterns, one for each field populated by the bridge.

⑤ Be careful: the search predicate factory expects paths relative to the object field where the named predicate was registered. Here the path `departmentCode` will be understood as `<path to the object field>.departmentCode`. See also Field paths.

⑥ Do not forget to call `toPredicate()` to return a `SearchPredicate` instance.

```
    @Entity
```

```
@Indexed
public class ItemStock {

    @Id
    @PropertyBinding(binder = @PropertyBinderRef(type = SkuIdentifierBinder.class)) ①
    private String skuId;

    private int amountInStock;

    // Getters and setters
    // ...


}
```

① Apply the bridge using the `@PropertyBinding` annotation. The predicate will be available in the Search DSL, as shown in Named: call a predicate defined in the mapping.

12.11. Assigning default bridges with the bridge resolver

12.11.1. Basics

Both the `@*Field` annotations and the `@DocumentId` annotation support a broad range of standard types by default, without needing to tell Hibernate Search how to convert values to something that can be indexed.

Under the hood, the support for default types is handled by the bridge resolver. For example, when a property is mapped with `@GenericField` and neither `@GenericField.valueBridge` nor `@GenericField.valueBinder` is set, Hibernate Search will resolve the type of this property, then pass it to the bridge resolver, which will return an appropriate bridge, or fail if there isn't any.

It is possible to customize the bridge resolver, to override existing default bridges (indexing `java.util.Date` differently, for example) or to define default bridges for additional types (a geospatial type from an external library, for example).

To that end, define a mapping configurer as explained in Programmatic mapping, then define bridges as shown below:

Example 12.52: Defining default bridges with a mapping configurer

```
public class MyDefaultBridgesConfigurer implements HibernateOrmSearchMappingConfigurer {
    @Override
    public void configure(HibernateOrmMappingConfigurationContext context) {
        context.bridges().exactType( MyCoordinates.class )
                .valueBridge( new MyCoordinatesBridge() ); ①

        context.bridges().exactType( MyProductId.class )
                .identifierBridge( new MyProductIdBridge() ); ②

        context.bridges().exactType( ISBN.class )
                .valueBinder( new ValueBinder() { ③
                    @Override
                    public void bind(ValueBindingContext<?> context) {
                        context.bridge( ISBN.class, new ISBNValueBridge(),
                                context.typeFactory().asString().normalizer( "isbn" ) );
                    }
                } );
    }
}
```

① Use our custom bridge (`MyCoordinatesBridge`) by default when a property of type `MyCoordinates` is mapped to an index field (e.g. with `@GenericField`).

② Use our custom bridge (`MyProductBridge`) by default when a property of type `MyProductId` is mapped to a document identifier (e.g. with `@DocumentId`).

③ It's also possible to specify a binder instead of a bridge, so that additional settings can be tuned. Here we're assigning the "isbn" normalizer every time we map an ISBN to an index field.

12.11.2. Assigning a single binder to multiple types

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

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

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

For more advanced use cases, it is also possible to assign a single binder to subtypes of a given type. This is useful when many types should be indexed similarly.

Below is an example where enums are not indexed as their `.name()` (which is the default), but instead are indexed as their label retrieved from an external service.

Example 12.53: Assigning a single default binder to multiple types with a mapping configurer

```
context.bridges().subTypesOf( Enum.class ) ①
        .valueBinder( new ValueBinder() {
            @Override
            public void bind(ValueBindingContext<?> context) {
                Class<?> enumType = context.bridgedElement().rawType(); ②
                doBind( context, enumType );
            }

            private <T> void doBind(ValueBindingContext<?> context, Class<T> enumType) {
                BeanHolder<EnumLabelService> serviceHolder = context.beanResolver()
                        .resolve( EnumLabelService.class, BeanRetrieval.ANY ); ③
                context.bridge(
                        enumType,
                        new EnumLabelBridge<>( enumType, serviceHolder )
                ); ④
            }
        } );
```

① Match all subtypes of `Enum`.
② Retrieve the type of the element being bridged.
③ Retrieve an external service (through CDI/Spring).
④ Create and assign the bridge.

12.12. Projection binder

12.12.1. Basics

Projection binders are an advanced feature that application developers generally shouldn't need to bother with. Before resorting to custom projection binders, consider relying on explicit projection constructor parameter mapping using built-in annotations such as `@IdProjection`, `@FieldProjection`, `@ObjectProjection`, …

A projection binder is a pluggable component that implements the binding of a constructor parameter to a projection. It is applied to a parameter of a projection constructor with the `@ProjectionBinding` annotation or with a custom annotation.

The projection binder can inspect the constructor parameter, and is expected to assign a projection definition to that constructor parameter, so that whenever the projection constructor is invoked, Hibernate Search will pass the result of that projection through that constructor parameter.

Implementing a projection binder requires two components:

1. A custom implementation of `ProjectionBinder`, to bind the projection definition to the parameter at bootstrap. This involves inspecting the constructor parameter if necessary, and instantiating the projection definition.
2. A custom implementation of `ProjectionDefinition`, to instantiate the projection at runtime. This involves using the projection DSL and returning the resulting `SearchProjection`.

Below is an example of a custom projection binder that binds a parameter of type `String` to a projection to the `title` field in the index.

A similar result can be achieved without a custom projection binder. This is just to keep the example simple.

Example 12.54: Implementing and using a `ProjectionBinder`

```
public class MyFieldProjectionBinder implements ProjectionBinder { ①
    @Override
    public void bind(ProjectionBindingContext context) { ②
        context.definition( ③
                String.class, ④
                new MyProjectionDefinition() ⑤
        );
    }

    // ... class continues below
```

① The binder must implement the `ProjectionBinder` interface.
② Implement the `bind` method in the binder.
③ Call `context.definition(…)` to define the projection to use.
④ Pass the expected type of the constructor parameter.
⑤ Pass the projection definition instance, which will create the projection at runtime.

```
    // ... class MyFieldProjectionBinder (continued)

    private static class MyProjectionDefinition ①
            implements ProjectionDefinition<String> { ②
        @Override
        public SearchProjection<String> create(ProjectionDefinitionContext context) {
            return context.projection().field( "title", String.class ) ③
                    .toProjection(); ④
        }
    }
}
```

① Here the definition class is nested in the binder class, because it is more convenient, but you are obviously free to implement it as you wish: as a lambda expression, in a separate Java file…
② The definition must implement the `ProjectionDefinition` interface. One generic type argument must be provided: the type of the projected value, i.e. the type of the constructor parameter.
③ Use the provided `SearchProjectionFactory` and the projection DSL to define the appropriate projection.
④ Get the resulting projection by calling `.toProjection()` and return it.

```
@ProjectionConstructor
public record MyBookProjection(
        @ProjectionBinding(binder = @ProjectionBinderRef( ①
                type = MyFieldProjectionBinder.class
        ))
        String title) {
}
```

① Apply the binder using the `@ProjectionBinding` annotation.

The book projection can then be used as any custom projection type, and its `title` parameter will be initialized with values returned by the custom projection definition:

```
List<MyBookProjection> hits = searchSession.search( Book.class )
        .select( MyBookProjection.class )
        .where( f -> f.matchAll() )
        .fetchHits( 20 );
```

12.12.2. Multi-valued projections

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

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

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

You can call `.containerElement()` on the context passed to the projection binder in order to discover whether the constructor parameter being bound is some sort of container wrapping the value/values (according to the same rules as implicit inner projection inference). If the returned value is a non-empty optional, then the `.constructorParameter()` will provide access to the container type in use. Additionally, the same context has access to a factory (`.projectionCollectorProviderFactory()`) from which a projection collector can be obtained based on a container and element types (if the container type is `null` then the factory will return the `nullable()` single-valued collector).

Example 12.55: Implementing and using a `ProjectionBinder` supporting multi-valued projections

```
public class MyFieldProjectionBinder implements ProjectionBinder {
    @Override
    public void bind(ProjectionBindingContext context) {
        Optional<PojoModelValue<?>> containerElement = context.containerElement(); ①
        if ( containerElement.isPresent() ) {
            context.definition( String.class, new MyProjectionDefinition() ); ②
        }
        else {
            throw new RuntimeException( "This binder only supports container-wrapped constructor parameters" );
③
        }
    }

    private static class MyProjectionDefinition
            implements ProjectionDefinition<List<String>> { ④
        @Override
        public SearchProjection<List<String>> create(ProjectionDefinitionContext context) {
            return context.projection().field( "tags", String.class )
                    .collector( ProjectionCollector.list() ) ④
                    .toProjection();
        }
    }
}
```

① `containerElement()` returns an optional that contains a type of the container elements if and only if the constructor parameter is considered as such that is wrapped in a container, e.g. a multivalued collection.

② Call `context.definition(…)` to define the projection to use.

③ Here we're failing for single-valued (nullable) constructor parameters, but we could theoretically fall back to a single-valued projection using the `ProjectionCollector.nullable()`.

④ The projection definition, being multivalued, must implement `ProjectionDefinition<SomeCollection<T>>`, where `T` is the expected type of projected values, `SomeCollection` is one of the supported collection types available in `ProjectionCollector`, and must configure returned projections accordingly. If the required collection type is not present in the `ProjectionCollector`, then a custom projection collector provider can be supplied.

```
@ProjectionConstructor
public record MyBookProjection(
        @ProjectionBinding(binder = @ProjectionBinderRef( ①
                type = MyFieldProjectionBinder.class
        ))
        List<String> tags) {
}
```

① Apply the binder using the `@ProjectionBinding` annotation.

The book projection can then be used as any custom projection type, and its `tags` parameter will be initialized with values returned by the custom projection definition:

```
List<MyBookProjection> hits = searchSession.search( Book.class )
        .select( MyBookProjection.class )
        .where( f -> f.matchAll() )
        .fetchHits( 20 );
```

12.12.3. Composing projection constructors

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

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

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

You can call `.createObjectDefinition( "someFieldPath", SomeType.class )` on the context passed to the projection binder in order to retrieve the definition of an object projection based on the projection constructor mapping of `SomeType`.

This effectively allows using projection constructors within projection binders, simply by passing the resulting definition to `.definition(…)` or by delegating to it in a custom projection definition.

Other methods exposed on the binding context work similarly:

- `.createObjectDefinition(…, ProjectionCollector.Provider)` returns an object projection definition with the provided collector applied and can be used for multivalued object projections or object projections wrapped in some other containers.
- `.createCompositeDefinition(…)` returns a (single-valued) composite projection definition (which, on contrary to an object projection, is not bound to an object field in the index).

Below is an example using `.createObjectDefinition(…)` to delegate to another projection constructor.

A similar result can be achieved without a custom projection binder, simply by relying on implicit inner projection inference or by using `@ObjectProjection`. This is just to keep the example simple.

Example 12.56: Implementing and using a `ProjectionBinder` that delegates to a projection constructor

```
public class MyObjectFieldProjectionBinder implements ProjectionBinder {
    @Override
    public void bind(ProjectionBindingContext context) {
        var authorProjection = context.createObjectDefinition( ①
                "author", ②
                MyBookProjection.MyAuthorProjection.class, ③
                TreeFilterDefinition.includeAll() ④
        );
        context.definition( ⑤
                MyBookProjection.MyAuthorProjection.class,
                authorProjection
        );
    }
}
```

① Call `createObjectDefinition(…)` to create a definition to delegate to.
② Pass the name of the object field to project on.
③ Pass the projected type.
④ Pass the filter for nested projections; here we're not filtering at all. This controls the same feature as `includePaths`/`excludePaths`/`includeDepths` in `@ObjectProjection`.
⑤ Call `definition(…)` and pass the definition created just above.

```
@ProjectionConstructor
public record MyBookProjection(
        @ProjectionBinding(binder = @ProjectionBinderRef( ①
                type = MyObjectFieldProjectionBinder.class
        ))
        MyAuthorProjection author) {

    @ProjectionConstructor ②
    public record MyAuthorProjection(String name) {
    }
}
```

① Apply the binder using the `@ProjectionBinding` annotation.
② Make sure the projected type passed to `createObjectDefinition(…)` has a projection constructor.

The book projection can then be used as any custom projection type, and its `author` parameter will be initialized with values returned by the custom projection definition:

```
List<MyBookProjection> hits = searchSession.search( Book.class )
        .select( MyBookProjection.class )
        .where( f -> f.matchAll() )
        .fetchHits( 20 );
```

12.12.4. Passing parameters

There are two ways to pass parameters to property bridges:

- One is (mostly) limited to string parameters, but is trivial to implement.
- The other can allow any type of parameters, but requires you to declare your own annotations.

Simple, string parameters

You can pass string parameters to the `@ProjectionBinderRef` annotation and then use them later in the binder:

Example 12.57: Passing parameters to a `ProjectionBinder` using the `@ProjectionBinderRef` annotation

```
public class MyFieldProjectionBinder implements ProjectionBinder {
    @Override
    public void bind(ProjectionBindingContext context) {
        String fieldName = context.param( "fieldName", String.class ); ①
        context.definition(
                String.class,
                new MyProjectionDefinition( fieldName ) ②
        );
    }

    private static class MyProjectionDefinition
            implements ProjectionDefinition<String> {

        private final String fieldName;

        public MyProjectionDefinition(String fieldName) { ②
            this.fieldName = fieldName;
        }

        @Override
        public SearchProjection<String> create(ProjectionDefinitionContext context) {
            return context.projection().field( fieldName, String.class ) ③
                    .toProjection();
        }
    }
}
```

① Use the binding context to get the parameter value.

The `param` method will throw an exception if the parameter has not been defined. Alternatively, use `paramOptional` to get an `java.util.Optional` that will be empty if the parameter has not been defined.

② Pass the parameter value as an argument to the definition constructor.

③ Use the parameter value in the projection definition.

```
@ProjectionConstructor
public record MyBookProjection(
        @ProjectionBinding(binder = @ProjectionBinderRef(
                type = MyFieldProjectionBinder.class,
                params = @Param(name = "fieldName", value = "title")
        )) String title) { ①
}
```

① Define the binder to use on the constructor parameter, setting the `fieldName` parameter.

Parameters with custom annotations

You can pass parameters of any type to the bridge by defining a custom annotation with attributes:

Example 12.58: Passing parameters to a `PropertyBinder` using a custom annotation

```
@Retention(RetentionPolicy.RUNTIME) ①
@Target({ ElementType.PARAMETER }) ②
@MethodParameterMapping(processor = @MethodParameterMappingAnnotationProcessorRef( ③
        type = MyFieldProjectionBinding.Processor.class
))
@Documented ④
public @interface MyFieldProjectionBinding {

    String fieldName() default ""; ⑤

    class Processor ⑥
            implements MethodParameterMappingAnnotationProcessor<MyFieldProjectionBinding> { ⑦
        @Override
        public void process(MethodParameterMappingStep mapping, MyFieldProjectionBinding annotation,
                MethodParameterMappingAnnotationProcessorContext context) {
            MyFieldProjectionBinder binder = new MyFieldProjectionBinder(); ⑧
            if ( !annotation.fieldName().isEmpty() ) { ⑨
                binder.fieldName( annotation.fieldName() );
            }
            mapping.projection( binder ); ⑩
        }
    }
}
```

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

② Since we will be mapping a projection definition to a projection constructor, allow the annotation to target method parameters (constructors are methods).

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

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

⑤ Define an attribute of type `String` to specify the field name.

⑥ Here the processor class is nested in the annotation class, because it is more convenient, but you are obviously free to implement it in a separate Java file.

⑦ The processor must implement the `MethodParameterMappingAnnotationProcessor` interface, setting its generic type argument to the type of the corresponding annotation.

⑧ In the annotation processor, instantiate the binder.

⑨ Process the annotation attributes and pass the data to the binder.

Here we're using a setter, but passing the data through the constructor would work, too.

⑩ Apply the binder to the constructor parameter.

```
public class MyFieldProjectionBinder implements ProjectionBinder {

    private String fieldName = "name";

    public MyFieldProjectionBinder fieldName(String fieldName) { ①
        this.fieldName = fieldName;
        return this;
    }

    @Override
    public void bind(ProjectionBindingContext context) {
        context.definition(
                String.class,
                new MyProjectionDefinition( fieldName ) ②
        );
    }

    private static class MyProjectionDefinition
            implements ProjectionDefinition<String> {
```

```
        private final String fieldName;

        public MyProjectionDefinition(String fieldName) { ②
            this.fieldName = fieldName;
        }

        @Override
        public SearchProjection<String> create(ProjectionDefinitionContext context) {
            return context.projection().field( fieldName, String.class ) ③
                    .toProjection();
        }
    }
}
```

① Implement setters in the binder. Alternatively, we could expose a parameterized constructor.
② In the `bind` method, use the value of parameters. Here we pass the parameter value as an argument to the definition constructor.
③ Use the parameter value in the projection definition.

```
@ProjectionConstructor
public record MyBookProjection(
        @MyFieldProjectionBinding(fieldName = "title") ①
        String title) {
}
```

① Apply the binder using its custom annotation, setting the `fieldName` parameter.

12.12.5. Injecting beans into the binder

With compatible frameworks, Hibernate Search supports injecting beans into:

- the `MethodParameterMappingAnnotationProcessor` if you use custom annotations.
- the `ProjectionBinder` if you use the `@ProjectionBinding` annotation.

This only applies to beans instantiated through Hibernate Search's bean resolution. As a rule of thumb, if you need to call `new MyBinder()` explicitly at some point, the binder won't get auto-magically injected.

The context passed to the property binder's `bind` method also exposes a `beanResolver()` method to access the bean resolver and instantiate beans explicitly.

See Bean injection for more details.

12.12.6. Programmatic mapping

You can apply a projection binder through the programmatic mapping too. Just pass an instance of the binder to `.projection(…)`. You can pass arguments either through the binder's constructor, or through setters.

Example 12.59: Applying a `ProjectionBinder` with `.projection(…)`

```
TypeMappingStep myBookProjectionMapping = mapping.type( MyBookProjection.class );
myBookProjectionMapping.mainConstructor().projectionConstructor();
myBookProjectionMapping.mainConstructor().parameter( 0 )
        .projection( new MyFieldProjectionBinder().fieldName( "title" ) );
```

12.12.7. Other incubating features

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

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

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

The context passed to the projection binder's `bind` method exposes a `constructorParameter()` method that gives access to metadata about the constructor parameter being bound.

The metadata can be used to inspect the constructor parameter in details:

- Getting the name of the constructor parameter.
- Checking the type of the constructor parameter.

Similarly, the context used for multi-valued projection binding exposes a `containerElement()` method that gives access to the type of elements of the (multi-valued) constructor parameter type.

See the javadoc for more information.

The name of the constructor parameter is only available:

- For the canonical constructor of record types, regardless of compiler flags.
- For constructors of non-record types or non-canonical constructors of record types if and only if the type was compiled with the `-parameters` compiler flag.

Below is an example of the simplest use of this metadata, getting the constructor parameter name and using it as a field name.

Example 12.60: Projecting on a field whose name is the same as the constructor parameter name in a `ProjectionBinder`

```
public class MyFieldProjectionBinder implements ProjectionBinder {
    @Override
    public void bind(ProjectionBindingContext context) {
        var constructorParam = context.constructorParameter(); ①
        context.definition(
                String.class,
                new MyProjectionDefinition( constructorParam.name().orElseThrow() ) ②
        );
    }

    private static class MyProjectionDefinition
            implements ProjectionDefinition<String> {
        private final String fieldName;

        private MyProjectionDefinition(String fieldName) {
            this.fieldName = fieldName;
        }

        @Override
        public SearchProjection<String> create(ProjectionDefinitionContext context) {
            return context.projection().field( fieldName, String.class ) ③
                    .toProjection();
        }
    }
}
```

① Use the binding context to get the constructor parameter.
② Pass the name of the constructor parameter to the projection definition.
③ Use the name of the constructor parameter as the projected field name.

```
@ProjectionConstructor
public record MyBookProjection(
        @ProjectionBinding(binder = @ProjectionBinderRef( ①
                type = MyFieldProjectionBinder.class
        ))
        String title) {
}
```

① Apply the binder using the `@ProjectionBinding` annotation.

Chapter 13. Managing the index schema

13.1. Basics

Before indexes can be used for indexing or searching, they must be created on disk (Lucene) or in the remote cluster (Elasticsearch). With Elasticsearch in particular, this creation may not be obvious since it requires to describe the schema for each index, which includes in particular:

- the definition of every analyzer or normalizer used in this index;
- the definition of every single field used in this index, including in particular its type, the analyzer assigned to it, whether it requires doc values, etc.

Hibernate Search has all the necessary information to generate this schema automatically, so it is possible to delegate the task of managing the schema to Hibernate Search.

13.2. Automatic schema management on startup/shutdown

The property `hibernate.search.schema_management.strategy` can be set to one of the following values in order to define what to do with the indexes and their schema on startup and shutdown.

Strategy	Definition	Warnings
`none`	A strategy that does not do anything on startup or shutdown. Indexes and their schema will not be created nor deleted on startup or shutdown. Hibernate Search will **not even check** that the index actually exists.	With Elasticsearch, indexes and their schema will have to be created explicitly before startup.
`validate`	A strategy that does not change indexes nor their schema, but checks that indexes exist and validates their schema on startup. An exception will be thrown on startup if: • Indexes are missing • OR, with the Elasticsearch backend only, indexes exist but their schema does not match the requirements of the Hibernate Search mapping: missing fields, fields with incorrect type, missing analyzer definitions or normalizer definitions, … "Compatible" differences such as extra fields are ignored.	Indexes and their schema will have to be created explicitly before startup. With the Lucene backend, validation is limited to checking that the indexes exist, because local Lucene indexes don't have a schema.
`create`	A strategy that creates missing indexes and their schema on startup, but does not touch existing indexes and assumes their schema is correct without validating it.	Creating a schema does not populate indexed data.
`create-or-validate` (**default**)	A strategy that creates missing indexes and their schema on startup, and validates the schema of existing indexes. With the Elasticsearch backend only, an exception will be thrown on startup if some indexes already exist but their schema does not match the requirements of the Hibernate Search mapping: missing fields, fields with incorrect type, missing analyzer definitions or normalizer definitions, … "Compatible" differences such as extra fields are ignored.	Creating a schema does not populate indexed data. With the Lucene backend, validation is limited to checking that the indexes exist, because local Lucene indexes don't have a schema.

Strategy	Definition	Warnings
`create-or-update`	A strategy that creates missing indexes and their schema on startup, and updates the schema of existing indexes if possible.	Updating a schema does not update indexed data. **This strategy is unfit for production environments**, due to several limitations including the impossibility to change the type of an existing field or the requirement to close indexes while updating analyzer definitions (which is not possible at all on AWS). With the Lucene backend, schema update is a no-op, because local Lucene indexes don't have a schema.
`drop-and-create`	A strategy that drops existing indexes and re-creates them and their schema on startup.	All indexed data will be lost on startup.
`drop-and-create-and-drop`	A strategy that drops existing indexes and re-creates them and their schema on startup, then drops the indexes on shutdown.	All indexed data will be lost on startup and shutdown.

13.3. Manual schema management

Schema management does not have to happen automatically on startup and shutdown.

Using the `SearchSchemaManager` interface, it is possible to trigger schema management operations explicitly after Hibernate Search has started.

The most common use case is to set the automatic schema management strategy to `none` and handle the creation/deletion of indexes manually when some other conditions are met, for example the Elasticsearch cluster has finished booting.

After schema management operations are complete, you will often want to populate indexes. To that end, use the mass indexer.

The `SearchSchemaManager` interface exposes the following methods.

Method	Definition	Warnings
`validate()`	Does not change indexes nor their schema, but checks that indexes exist and validates their schema.	With the Lucene backend, validation is limited to checking that the indexes exist, because local Lucene indexes don't have a schema.
`createIfMissing()`	Creates missing indexes and their schema, but does not touch existing indexes and assumes their schema is correct without validating it.	Creating a schema does not populate indexed data.
`createOrValidate()`	Creates missing indexes and their schema, and validates the schema of existing indexes.	Creating a schema does not populate indexed data. With the Lucene backend, validation is limited to checking that the indexes exist, because local Lucene indexes don't have a schema.

Method	Definition	Warnings
`createOrUpdate()`	Creates missing indexes and their schema, and updates the schema of existing indexes if possible.	Updating a schema does not update indexed data. With the Elasticsearch backend, updating a schema may fail. With the Elasticsearch backend, updating a schema may close indexes while updating analyzer definitions (which is not possible at all on Amazon OpenSearch Service). With the Lucene backend, schema update is a no-op, because local Lucene indexes don't have a schema. (it just creates missing indexes).
`dropIfExisting()`	Drops existing indexes.	All indexed data will be lost.
`dropAndCreate()`	Drops existing indexes and re-creates them and their schema.	All indexed data will be lost.

Below is an example using a `SearchSchemaManager` to drop and create indexes, then using a mass indexer to re-populate the indexes. The `dropAndCreateSchemaOnStart` setting of the mass indexer would be an alternative solution to achieve the same results.

Example 13.1: Reinitializing indexes using a `SearchSchemaManager`

```
SearchSession searchSession = /* ... */ ①
SearchSchemaManager schemaManager = searchSession.schemaManager(); ②
schemaManager.dropAndCreate(); ③
searchSession.massIndexer()
        .purgeAllOnStart( false )
        .startAndWait(); ④
```

① Retrieve the `SearchSession`.
② Get a schema manager.
③ Drop and create the indexes. This method is synchronous and will only return after the operation is complete.
④ Optionally, trigger mass indexing.

You can also select entity types when creating a schema manager, to manage the indexes of these types only (and their indexed subtypes, if any):

Example 13.2: Reinitializing only some indexes using a `SearchSchemaManager`

```
SearchSchemaManager schemaManager = searchSession.schemaManager( Book.class ); ①
schemaManager.dropAndCreate(); ②
```

① Get a schema manager targeting the index mapped to the `Book` entity type.
② Drop and create the index for the `Book` entity only. Other indexes are unaffected.

13.4. How schema management works

Creating/updating a schema does not create/update indexed data

Creating or updating indexes and their schema through schema management will not populate the indexes:

- newly created indexes will always be empty.
- indexes with a recently updated schema will still contain the same indexed data, i.e. new fields won't be added to documents just because they were added to the schema.

This is by design: reindexing is a potentially long-running task that should be triggered explicitly. To populate indexes with pre-existing data from the database, use mass indexing.

Dropping the schema means losing indexed data

Dropping a schema will drop the whole index, including all indexed data.

A dropped index will need to be re-created through schema management, then populated with pre-existing data from the database through mass indexing.

Schema validation and update are not effective with Lucene

The Lucene backend will only validate that the index actually exists and create missing indexes, because there is no concept of schema in Lucene beyond the existence of index segments.

Schema validation is permissive

With Elasticsearch, schema validation is as permissive as possible:

- Fields that are unknown to Hibernate Search will be ignored.
- Settings that are more powerful than required will be deemed valid. For example, a field that is not marked as sortable in Hibernate Search but marked as `"docvalues": true` in Elasticsearch will be deemed valid.
- Analyzer/normalizer definitions that are unknown to Hibernate Search will be ignored.

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

Schema updates may fail

A schema update, triggered by the `create-or-update` strategy, may simply fail. This is because schemas may change in an incompatible way, such as a field having its type changed, or its analyzer changed, etc.

Worse, since updates are handled on a per-index basis, a schema update may succeed for one index but fail on another, leaving your schema as a whole half-updated.

For these reasons, **using schema updates in a production environment is not recommended**. Whenever the schema changes, you should either:

- drop and create indexes, then reindex.
- OR update the schema manually through custom scripts.

In this case, the `create-or-update` strategy will prevent Hibernate Search from starting, but it may already have successfully updated the schema for another index, making a rollback difficult.

Schema updates on Elasticsearch may close indexes

Elasticsearch does not allow updating analyzer/normalizer definitions on an open index. Thus, when analyzer or normalizer definitions have to be updated during a schema update, Hibernate Search will temporarily stop the affected indexes.

For this reason, the `create-or-update` strategy should be used with caution when multiple clients use Elasticsearch indexes managed by Hibernate Search: those clients should be synchronized in such a way that while Hibernate Search is starting, no other client needs to access the index.

Also, on Amazon OpenSearch Service running Elasticsearch (not OpenSearch) in version 7.1 or older, as well as on Amazon OpenSearch Serverless, the `_close`/`_open` operations are not supported, so **the schema update will fail** when trying to update analyzer definitions. The only workaround is to avoid the schema update on these platforms. It should be avoided in production environments regardless: see [schema-management-concepts-update-failure].

13.5. Exporting the schema

13.5.1. Exporting the schema to a set of files

The schema manager provides a way to export schemas to the filesystem. The output is backend-specific.

Schema exports are constructed based on the mapping information and configurations (e.g. such as the backend version). Resulting exports are not compared to or validated against the actual backend schema.

For Elasticsearch, the files provide the necessary information to create indexes (along with their settings and mapping). The file tree structure of an export is shown below:

```
# For the default backend the index schema will be written to:
.../backend/indexes/<index-name>/create-index.json
.../backend/indexes/<index-name>/create-index-query-params.json
# For additional named backends:
.../backends/<name of a particular backend>/indexes/<index-name>/create-index.json
.../backends/<name of a particular backend>/indexes/<index-name>/create-index-query-params.json
```

Example 13.3: Exporting the schema to the filesystem

```
SearchSchemaManager schemaManager = searchSession.schemaManager(); ①
schemaManager.exportExpectedSchema( targetDirectory ); ②
```

① Retrieve the `SearchSchemaManager` from a `SearchSession`.
② Export the schema to a target directory.

13.5.2. Exporting to a custom collector

Search schema managers allow walking through the schema exports based on the data such managers contains. To do so a `SearchSchemaCollector` must be implemented and passed to the schema manager's `exportExpectedSchema(..)` method.

Schema exports are constructed based on the mapping information and configurations (e.g. such as the backend version). Resulting exports are not compared to or validated against the actual backend schema.

Example 13.4: Exporting to a custom collector

```
SearchSchemaManager schemaManager = searchSession.schemaManager(); ①
schemaManager.exportExpectedSchema(
        new SearchSchemaCollector() { ②
            @Override
            public void indexSchema(Optional<String> backendName, String indexName, SchemaExport export) {
                String name = backendName.orElse( "default" ) + ":" + indexName; ③
                // perform any other actions with an index schema export
            }
        }
);
```

① Retrieve the `SearchSchemaManager` from a `SearchSession`.
② Instantiate and pass the `SearchSchemaCollector` to walk a schema.
③ Create a name from an index and backend names.

To access backend-specific functionality, an extension to `SchemaExport` can be applied:

```
new SearchSchemaCollector() {

    @Override
    public void indexSchema(Optional<String> backendName, String indexName, SchemaExport export)
{
        List<JsonObject> bodyParts = export
                .extension( ElasticsearchExtension.get() ) ①
                .bodyParts(); ②
    }
}
```

① Extend the `SchemaExport` with the Elasticsearch extension.
② Access an HTTP body of a request that is needed to create an index in an Elasticsearch cluster.

13.5.3. Exporting in offline mode

Sometimes it can be useful to export the schema offline, from an environment that doesn't have access to e.g. the Elasticsearch cluster.

See this section for more information on how to achieve offline startup.

Chapter 14. Indexing entities

14.1. Basics

There are multiple ways to index entities in Hibernate Search.

If you want to get to know the most popular ones, head directly to the following section:

- To keep indexes synchronized transparently as entities change in a Hibernate ORM `Session`, see listener-triggered indexing.
- To index a large amount of data — for example the whole database, when adding Hibernate Search to an existing application — see the `MassIndexer`.

Otherwise, the following table may help you figure out what's best for your use case.

Table 14.1: Comparison of indexing methods

Name and link	Use case	API	Mapper
Listener-triggered indexing	Handle incremental changes in application transactions	None: works implicitly without API calls	Hibernate ORM integration only
`MassIndexer`	Reindex large volumes of data in batches	Specific to Hibernate Search	Hibernate ORM integration or Standalone POJO Mapper
Jakarta Batch mass indexing job		Jakarta EE standard	Hibernate ORM integration only
Explicit indexing	Anything else	Specific to Hibernate Search	Hibernate ORM integration or Standalone POJO Mapper

14.2. Indexing plans

14.2.1. Basics

For listener-triggered indexing as well as some forms of explicit indexing, Hibernate Search relies on an "indexing plan" to aggregate "entity change" events and infer the resulting indexing operations to execute.

Indexing plans are not used for the `MassIndexer` or the Jakarta Batch mass indexing job: those assume all entities they process need to be indexed and don't need the more subtle mechanisms of indexing plans.

Here is how indexing plans work at a high level:

1. While the application performs entity changes, entity change events (entity created, updated, deleted) are added to the plan.

   For listener-triggered indexing (Hibernate ORM integration only) this happens implicitly as changes are performed, but it can also be done explicitly.

2. Eventually, the application decides changes are complete, and the plan processes change events added so far, either inferring which entities need to be reindexed and building the corresponding documents (no coordination) or building events to be sent to the outbox (`outbox-polling` coordination).

   For the Hibernate ORM integration this happens when the Hibernate ORM `Session` gets flushed (explicitly or as part of a transaction commit), while for the Standalone POJO Mapper this happens when the `SearchSession` is closed.

3. Finally the plan gets executed, triggering indexing, potentially asynchronously.

   For the Hibernate ORM integration this happens on transaction commit, while for the Standalone POJO Mapper this happens when the `SearchSession` is closed.

Below is a summary of key characteristics of indexing plans and how they vary depending on the configured coordination strategy.

Table 14.2: Comparison of indexing plans depending on the coordination strategy

Coordination strategy	**No coordination** (default)	**Outbox polling** (**Hibernate ORM integration only**)
Guarantee of indexes updates	Non-transactional, after the database transaction / `SearchSession.close()` returns	Transactional, on database transaction commit

Coordination strategy	No coordination (default)	Outbox polling (Hibernate ORM integration only)
Visibility of index updates	Configurable: immediate (poor performance) or eventual	Eventual
Overhead for application threads	Low to medium	Very low
Overhead for the database (Hibernate ORM integration only)	Low	Low to medium

14.2.2. Synchronization with the indexes

Basics

For a preliminary introduction to writing to and reading from indexes in Hibernate Search, including in particular the concepts of *commit* and *refresh*, see Commit and refresh.

When using the `outbox-polling` coordination strategy, the actual indexing plan performing the index changes is created asynchronously in a background thread. Because of that, with that coordination strategy it does not make sense to set a non-default indexing plan synchronization strategy, and doing so will lead to an exception on startup.

When a transaction is committed (Hibernate ORM integration) or the `SearchSession` is closed (Standalone POJO Mapper), with default coordination settings, the execution of the indexing plan (implicit (listener-triggered) or explicit) can block the application thread until indexing reaches a certain level of completion.

There are two main reasons for blocking the thread:

1. **Indexed data safety**: if, once the database transaction completes, index data must be safely stored to disk, an index commit is necessary. Without it, index changes may only be safe after a few seconds, when a periodic index commit happens in the background.
2. **Real-time search queries**: if, once the database transaction completes (for the Hibernate ORM integration) or the `SearchSession`'s `close()` method returns (for the Standalone POJO Mapper), any search query must immediately take the index changes into account, an index refresh is necessary. Without it, index changes may only be visible after a few seconds, when a periodic index refresh happens in the background.

These two requirements are controlled by the *synchronization strategy*. The default strategy is defined by the configuration property `hibernate.search.indexing.plan.synchronization.strategy`. Below is a reference of all available strategies and their guarantees.

Strategy	Throughput	Guarantees when the application thread resumes		
		Changes applied (with or without commit)	Changes safe from crash/power loss (commit)	Changes visible on search (refresh)
`async`	Best	No guarantee	No guarantee	No guarantee
`write-sync` (**default**)	Medium	Guaranteed	Guaranteed	No guarantee
`read-sync`	Medium to worst	Guaranteed	No guarantee	Guaranteed
`sync`	Worst	Guaranteed	Guaranteed	Guaranteed

Depending on the backend and its configuration, the `sync` and `read-sync` strategies may lead to poor indexing throughput, because the backend may not be designed for frequent, on-demand index refreshes.

This is why this strategy is only recommended if you know your backend is designed for it, or for integration tests. In particular, the `sync` strategy will work fine with the default configuration of the Lucene backend, but will perform poorly with the Elasticsearch backend.

Indexing failures may be reported differently depending on the chosen strategy:

- Failure to extract data from entities:
  - Regardless of the strategy, throws an exception in the application thread.

- Failure to apply index changes (i.e. I/O operations on the index):
  - For strategies that apply changes immediately: throws an exception in the application thread.
  - For strategies that do **not** apply changes immediately: forwards the failure to the failure handler, which by default will simply log the failure.
- Failure to commit index changes:
  - For strategies that guarantee an index commit: throws an exception in the application thread.
  - For strategies that do **not** guarantee an index commit: forwards the failure to the failure handler, which by default will simply log the failure.

Per-session override

While the configuration property mentioned above defines a default, it is possible to override this default on a particular session by calling `SearchSession#indexingPlanSynchronizationStrategy(…)` and passing a different strategy.

The built-in strategies can be retrieved by calling:

- `IndexingPlanSynchronizationStrategy.async()`
- `IndexingPlanSynchronizationStrategy.writeSync()`
- `IndexingPlanSynchronizationStrategy.readSync()`
- or `IndexingPlanSynchronizationStrategy.sync()`

Example 14.1: Overriding the indexing plan synchronization strategy

```
SearchSession searchSession = /* ... */ ①
searchSession.indexingPlanSynchronizationStrategy(
        IndexingPlanSynchronizationStrategy.sync()
); ②

entityManager.getTransaction().begin();
try {
    Book book = entityManager.find( Book.class, 1 );
    book.setTitle( book.getTitle() + " (2nd edition)" ); ③
    entityManager.getTransaction().commit(); ④
}
catch (RuntimeException e) {
    entityManager.getTransaction().rollback();
}

List<Book> result = searchSession.search( Book.class )
        .where( f -> f.match().field( "title" ).matching( "2nd edition" ) )
        .fetchHits( 20 ); ⑤
```

① Retrieve the `SearchSession`, which by default uses the synchronization strategy configured in properties.
② Override the synchronization strategy.
③ Change an entity.
④ Commit the changes, triggering reindexing.
⑤ The overridden strategy guarantees that the modified book will be present in these results, even though the query was executed *just after* the transaction commit (here we're using the Hibernate ORM integration).

Custom strategy

You can also implement custom strategy. The custom strategy can then be set just like the built-in strategies:

- as the default by setting the configuration property `hibernate.search.indexing.plan.synchronization.strategy` to a bean reference pointing to the custom implementation, for example `class:com.mycompany.MySynchronizationStrategy`.
- at the session level by passing an instance of the custom implementation to `SearchSession#indexingPlanSynchronizationStrategy(…)`.

14.2.3. Indexing plan filter

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

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

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

In some scenarios, it might be helpful to pause the explicit and listener-triggered indexing programmatically, for example, when importing larger amounts of data. Hibernate Search allows configuring application-wide and session-level filters to manage which types are tracked for changes and indexed.

Example 14.2: Configuring an application-wide filter

```
SearchMapping searchMapping = /* ... */ ①
searchMapping.indexingPlanFilter( ②
        ctx -> ctx.exclude( EntityA.class ) ③
                .include( EntityExtendsA2.class )
);
```

Configuring an application-wide filter requires an instance of the `SearchMapping`.

① Retrieve the `SearchMapping`.
② Start the declaration of the indexing plan filter.
③ Configure included/excluded types through the `SearchIndexingPlanFilter`

Example 14.3: Configuring a session-level filter

```
SearchSession session = /* ... */ ①
session.indexingPlanFilter(
        ctx -> ctx.exclude( EntityA.class ) ②
                .include( EntityExtendsA2.class )
);
```

Configuring a session level filter is available through an instance of the `SearchSession`.

① Retrieve the `SearchSession`
② Configure included/excluded types through the `SearchIndexingPlanFilter`

Filter can be defined by providing indexed and contained types as well as their supertypes. Interfaces are not allowed and passing an interface class to any of the filter definition methods will result in an exception. If dynamic types represented by a `Map` are used then their names must be used to configure the filter. Filter rules are:

- If the type `A` is explicitly included by the filter, then a change to an object that is exactly of a type `A` is processed.
- If the type `A` is explicitly excluded by the filter, then a change to an object that is exactly of a type `A` is ignored.
- If the type `A` is explicitly included by the filter, then a change to an object that is exactly of a type `B`, which is a subtype of the type `A`, is processed unless the filter explicitly excludes a more specific supertype of a type `B`.
- If the type `A` is excluded by the filter explicitly, then a change to an object that is exactly of a type `B`, which is a subtype of type the `A`, is ignored unless the filter explicitly includes a more specific supertype of a type `B`.

A session-level filter takes precedence over an application-wide one. If the session-level filter configuration does not either explicitly or through inheritance include/exclude the exact type of an entity, then the decision will be made by the application-wide filter. If an application-wide filter also has no explicit configuration for a type, then this type is considered to be included.

In some cases we might need to disable the indexing entirely. Listing all entities one by one might be cumbersome, but since filter configuration is implicitly applied to subtypes, `.exclude(Object.class)` can be used to exclude all types. Conversely, `.include(Object.class)` can be used to enable indexing within a session filter when the application-wide filter disables indexing completely.

Example 14.4: Disable all indexing within a session

```
SearchSession searchSession = /* ... */ ①
searchSession.indexingPlanFilter(
        ctx -> ctx.exclude( Object.class ) ②
);
```

Configuring a session level filter is available through an instance of the `SearchSession`.

① Retrieve the `SearchSession`
② Excluding `Object.class` will lead to excluding all its subtypes which means nothing will be included.

Example 14.5: Enable indexing in the session while application-wide indexing is paused

```
SearchMapping searchMapping = /* ... */ ①
searchMapping.indexingPlanFilter(
        ctx -> ctx.exclude( Object.class ) ②
);
```

```
SearchSession searchSession = /* ... */ ③
searchSession.indexingPlanFilter(
        ctx -> ctx.include( Object.class ) ④
);
```

① Retrieve the `SearchMapping`.
② An application-wide filter disables any indexing
③ Retrieve the `SearchSession`
④ A session level filter re-enables indexing **for changes happening in current session only**

Trying to configure the same type as both included and excluded at the same time by the same filter will lead to an exception being thrown.

Only an application-wide filter is safe to use when using the `outbox-polling` coordination strategy. When this coordination strategy is in use, entities are loaded and indexed in a different session from the one where they were changed. It might lead to unexpected results as the session where events are processed will not apply the filter configured by the session in which entities were modified. An exception will be thrown if such a filter is configured unless this filter excludes all the types to prevent any unexpected consequences of configuring session-level filters with this coordination strategy.

14.3. Implicit, listener-triggered indexing

14.3.1. Basics

This feature is only available with the Hibernate ORM integration.

It **cannot** be used with the Standalone POJO Mapper in particular.

By default, every time an entity is changed through a Hibernate ORM Session, if the entity type is mapped to an index, Hibernate Search updates the relevant index transparently.

Here is how listener-triggered indexing works at a high level:

1. When the Hibernate ORM `Session` gets flushed (explicitly or as part of a transaction commit), Hibernate ORM determines what changed exactly (entity created, updated, deleted), forwards the information to Hibernate Search.
2. Hibernate Search adds this information to a (session-scoped) indexing plan and the plan processes change events added so far, either inferring which entities need to be reindexed and building the corresponding documents (no coordination) or building events to be sent to the outbox (`outbox-polling` coordination).
3. On database transaction commit, the plan gets executed, either sending the document indexing/deletion request to the backend (no coordination) or sending the events to the database (`outbox-polling` coordination).

Below is a summary of key characteristics of listener-triggered indexing and how they vary depending on the configured coordination strategy.

Follow the links for more details.

Table 14.4: Comparison of listener-triggered indexing depending on the coordination strategy

Coordination strategy	No coordination (default)	Outbox polling
Detects changes occurring in ORM sessions (`session.persist(…)`, `session.delete(…)`, setters, …)	Yes	
Detects changes caused by JPQL or SQL queries (`insert`/`update`/`delete`)	No	
Associations must be updated on both sides	Yes	
Changes triggering reindexing	Only relevant changes	
Guarantee of indexes updates	Non-transactional, after the database transaction / `SearchSession.close()` returns	Transactional, on database transaction commit
Visibility of index updates	Configurable: immediate (poor performance) or eventual	Eventual
Overhead for application threads	Low to medium	Very low
Overhead for the database	Low	Low to medium

14.3.2. Configuration

Listener-triggered indexing may be unnecessary if your index is read-only or if you update it regularly by reindexing, either using the `MassIndexer`, using the Jakarta Batch mass indexing job, or explicitly.

You can disable listener-triggered indexing by setting the configuration property `hibernate.search.indexing.listeners.enabled` to `false`.

As listener-triggered indexing uses indexing plans under the hood, several configuration options affecting indexing plans will affect listener-triggered indexing as well:

- The indexing plan synchronization strategy.
- The indexing plan filter.

14.3.3. In-session entity change detection and limitations

Hibernate Search uses internal events of Hibernate ORM in order to detect changes. These events will be triggered if you actually manipulate managed entity objects in your code: calls to `session.persist(…)`, `session.delete(…)`, to entity setters, etc.

This works great for most applications, but you need to consider some limitations:

- Listener-triggered indexing only considers changes applied directly to entity instances in Hibernate ORM sessions
- Listener-triggered indexing ignores asymmetric association updates

14.3.4. Dirty checking

Hibernate Search is aware of the entity properties that are accessed when building indexed documents. When processing Hibernate ORM entity change events, it is also aware of which properties actually changed. Thanks to that knowledge, it is able to detect which entity changes are actually relevant to indexing, and to skip reindexing when a property is modified, but does not affect the indexed document.

14.4. Indexing a large amount of data with the `MassIndexer`

14.4.1. Basics

There are cases where listener-triggered or explicit indexing is not enough, because pre-existing data has to be indexed:

- when restoring a database backup;
- when indexes had to be wiped, for example because the Hibernate Search mapping or some core settings changed;
- when entities cannot be indexed as they change (e.g. with listener-triggered indexing) for performance reasons, and periodic reindexing (every night, …) is preferred.

To address these situations, Hibernate Search provides the `MassIndexer`: a tool to rebuild indexes completely based on the content of an external datastore (for the Hibernate ORM integration, that datastore is the database). The `MassIndexer` can be told to reindex a few

selected indexed types, or all of them.

The `MassIndexer` takes the following approach to achieve a reasonably high throughput:

- Indexes are purged completely when mass indexing starts.
- Mass indexing is performed by several parallel threads, each loading data from the database and sending indexing requests to the indexes, not triggering any commit or refresh.
- An implicit flush (commit) and refresh are performed upon mass indexing completion, except for Amazon OpenSearch Serverless since it doesn't support explicit flushes or refreshes.

Because of the initial index purge, and because mass indexing is a very resource-intensive operation, it is recommended to take your application offline while the `MassIndexer` is working.

Querying the index while a `MassIndexer` is busy may be slower than usual and will likely return incomplete results.

The following snippet of code will rebuild the index of all indexed entities, deleting the index and then reloading all entities from the database.

Example 14.6: Reindexing everything using a `MassIndexer`

```
SearchSession searchSession = /* ... */ ①
searchSession.massIndexer() ②
        .startAndWait(); ③
```

① Retrieve the `SearchSession`.
② Create a `MassIndexer` targeting every indexed entity type.
③ Start the mass indexing process and return when it is over.

The `MassIndexer` creates its own, separate sessions and (read-only) transactions, so there is no need to begin a database transaction before the `MassIndexer` is started or to commit a transaction after it is done.

A note to MySQL users: the `MassIndexer` uses forward-only scrollable results to iterate on the primary keys to be loaded, but MySQL's JDBC driver will preload all values in memory.

To avoid this "optimization" set the `idFetchSize` parameter to `Integer.MIN_VALUE`.

Although the `MassIndexer` is simple to use, some tweaking is recommended to speed up the process. Several optional parameters are available, and can be set as shown below, before the mass indexer starts. See `MassIndexer` parameters for a reference of all available parameters, and Tuning the `MassIndexer` for best performance for details about key topics.

Example 14.7: Using a tuned MassIndexer

```
searchSession.massIndexer() ①
        .idFetchSize( 150 ) ②
        .batchSizeToLoadObjects( 25 ) ③
        .threadsToLoadObjects( 12 ) ④
        .startAndWait(); ⑤
```

① Create a `MassIndexer`.
② Load `Book` identifiers by batches of 150 elements.
③ Load `Book` entities to reindex by batches of 25 elements.
④ Create 12 parallel threads to load the `Book` entities.
⑤ Start the mass indexing process and return when it is over.

Running the `MassIndexer` with many threads may require many connections to the database. If you don't have a sufficiently large connection pool, the `MassIndexer` itself and/or your other applications could starve and be unable to serve other requests: make sure you size your connection pool according to the mass indexing parameters, as explained in Threads and connections.

14.4.2. Selecting types to be indexed

You can select entity types when creating a mass indexer, to reindex only these types (and their indexed subtypes, if any):

Example 14.8: Reindexing selected types using a `MassIndexer`

```
searchSession.massIndexer( Book.class ) ①
        .startAndWait(); ②
```

① Create a `MassIndexer` targeting the `Book` type and its indexed subtypes (if any).
② Start the mass indexing process for the selected types and return when it is over.

14.4.3. Mass indexing multiple tenants

Examples in sections above create a mass indexer from a given session, which will always limit mass indexing to the tenant targeted by that session.

When using multi-tenancy you can reindex multiple tenants at once by retrieving the mass indexer from a `SearchScope` and passing a collection of tenant identifiers:

Example 14.9: Reindexing multiple tenants listed explicitly using a `MassIndexer`

```
SearchMapping searchMapping = /* ... */ ①
searchMapping.scope( Object.class ) ②
        .massIndexer( asSet( "tenant1", "tenant2" ) ) ③
        .startAndWait(); ④
```

① Retrieve the `SearchMapping`.
② Retrieve a `SearchScope` targeting all types that we want reindexed; here we use `Object.class`, meaning "all indexed types that extent `Object`", i.e. simply all indexed types.
③ Pass the identifiers of tenants we want to mass index and create the mass indexer.
④ Start the mass indexing process and return when it is over.

With the Hibernate ORM mapper, if you included the comprehensive list of tenants in Hibernate Search's configuration, you can simply call `scope.massIndexer()` without any argument, and the resulting mass indexer will target all configured tenants:

Example 14.10: Reindexing multiple tenants configured implicitly using a `MassIndexer`

```
SearchMapping searchMapping = /* ... */ ①
searchMapping.scope( Object.class ) ②
        .massIndexer() ③
        .startAndWait(); ④
```

① Retrieve the `SearchMapping`.
② Retrieve a `SearchScope` targeting all types that we want reindexed; here we use `Object.class`, meaning "all indexed types that extent `Object`", i.e. simply all indexed types.
③ Create a mass indexer targeting all tenants included in the configuration.
④ Start the mass indexing process and return when it is over.

14.4.4. Running the mass indexer asynchronously

It is possible to run the mass indexer asynchronously, because it does not rely on the original Hibernate ORM session. When used asynchronously, the mass indexer will return a completion stage to track the completion of mass indexing:

Example 14.11: Reindexing asynchronously using a `MassIndexer`

```
searchSession.massIndexer() ①
        .start() ②
        .thenRun( () -> { ③
            TEST_LOGGER.info( "Mass indexing succeeded!" );
        } )
```

```
                .exceptionally( throwable -> {
                    TEST_LOGGER.error( "Mass indexing failed!", throwable );
                    return null;
                } );

// OR
Future<?> future = searchSession.massIndexer()
        .start()
        .toCompletableFuture(); ④
```

① Create a `MassIndexer`.
② Start the mass indexing process, but do not wait for the process to finish. A `CompletionStage` is returned.
③ The `CompletionStage` exposes methods to execute more code after indexing is complete.
④ Alternatively, call `toCompletableFuture()` on the returned object to get a `Future`.

14.4.5. Conditional reindexing

This feature is only available with the Hibernate ORM integration.

It **cannot** be used with the Standalone POJO Mapper in particular.

You can select a subset of target entities to be reindexed by passing a condition as string to the mass indexer. The condition will be applied when querying the database for entities to index.

The condition string is expected to follow the Hibernate Query Language (HQL) syntax. Accessible entity properties are those of the entity being reindexed (and nothing more).

Example 14.12: Use of conditional reindexing

```
SearchSession searchSession = /* ... */ ①
MassIndexer massIndexer = searchSession.massIndexer(); ②
massIndexer.type( Book.class ).reindexOnly( "publicationYear < 1950" ); ③
massIndexer.type( Author.class ).reindexOnly( "birthDate < :cutoff" ) ④
        .param( "cutoff", Year.of( 1950 ).atDay( 1 ) ); ⑤
massIndexer.startAndWait(); ⑥
```

① Retrieve the `SearchSession`.
② Create a `MassIndexer` targeting every indexed entity type.
③ Reindex only the books published before year 1950.
④ Reindex only the authors born prior to a given local date.
⑤ In this example the cutoff date is passed as a query parameter.
⑥ Start the mass indexing process and return when it is over.

Even if the reindexing is applied on a subset of entities, by default **all entities** will be purged at the start. The purge can be disabled completely, but when enabled there is no way to filter the entities that will be purged.

See HSEARCH-3304 for more information.

14.4.6. `MassIndexer` parameters

Table 14.5: `MassIndexer` parameters

Setter	Default value	Description
`typesToIndexInParallel(int)`	1	The number of types to index in parallel.
`threadsToLoadObjects(int)`	6	The number of threads for entity loading, for **each type indexed in parallel**. That is to say, the number of threads spawned for entity loading will be `typesToIndexInParallel * threadsToLoadObjects` (+ 1 thread per type to retrieve the IDs of entities to load).

Setter	Default value	Description
`idFetchSize(int)`	`100`	**Only supported with the Hibernate ORM integration.** The fetch size to be used when loading primary keys. Some databases accept special values, for example MySQL might benefit from using `Integer#MIN_VALUE`, otherwise it will attempt to preload everything in memory.
`batchSizeToLoadObjects(int)`	`10`	**Only supported with the Hibernate ORM integration.** The fetch size to be used when loading entities from database. Some databases accept special values, for example MySQL might benefit from using `Integer#MIN_VALUE`, otherwise it will attempt to preload everything in memory.
`dropAndCreateSchemaOnStart(boolean)`	`false`	Drops the indexes and their schema (if they exist) and re-creates them before indexing. Indexes will be unavailable for a short time during the dropping and re-creation, so this should only be used when failures of concurrent operations on the indexes (listener-triggered indexing, …) are acceptable. This should be used when the existing schema is known to be obsolete, for example when the Hibernate Search mapping changed and some fields now have a different type, a different analyzer, new capabilities (projectable, …), etc. This may also be used when the schema is up-to-date, since it can be faster than a purge (`purgeAllOnStart`) on large indexes, especially with the Elasticsearch backend. As an alternative to this parameter, you can also use a schema manager to manage schemas manually at the time of your choosing: Manual schema management.
`purgeAllOnStart(boolean)`	Default value depends on `dropAndCreateSchemaOnStart(boolean)`. Defaults to `false` if the mass indexer is configured to drop and create the schema on start, to `true` otherwise.	Removes all entities from the indexes before indexing. Only set this to `false` if you know the index is already empty; otherwise, you will end up with duplicates in the index.
`mergeSegmentsAfterPurge(boolean)`	`true` in general, `false` on Amazon OpenSearch Serverless	Force merging of each index into a single segment after the initial index purge, just before indexing. This setting has no effect if `purgeAllOnStart` is set to false.
`mergeSegmentsOnFinish(boolean)`	`false`	Force merging of each index into a single segment after indexing. This operation does not always improve performance: see Merging segments and performance.

Setter	Default value	Description
`cacheMode(CacheMode)`	`CacheMode.IGNORE`	**Only supported with the Hibernate ORM integration.** The Hibernate `CacheMode` when loading entities. The default is `CacheMode.IGNORE`, and it will be the most efficient choice in most cases, but using another mode such as `CacheMode.GET` may be more efficient if many of the entities being indexed refer to a small set of other entities.
`transactionTimeout`	-	**Only supported in JTA-enabled environments and with the Hibernate ORM integration.** Timeout of transactions for loading ids and entities to be re-indexed. The timeout should be long enough to load and index all entities of one type. Note that these transactions are read-only, so choosing a large value (e.g. `1800`, meaning 30 minutes) should not cause any problem.
`limitIndexedObjectsTo(long)`	-	**Only supported with the Hibernate ORM integration.** The maximum number of results to load per entity type. This parameter let you define a threshold value to avoid loading too many entities accidentally. The value defined must be greater than 0. The parameter is not used by default. It is equivalent to keyword `LIMIT` in SQL.
`monitor(MassIndexingMonitor)`	A logging monitor.	The component responsible for monitoring progress of mass indexing. As a `MassIndexer` can take some time to finish its job, it is often necessary to monitor its progress. The default, built-in monitor logs progress periodically at the `INFO` level, but a custom monitor can be set by implementing the `MassIndexingMonitor` interface and passing an instance using the `monitor` method. The built-in monitor's behaviour can be customized through `DefaultMassIndexingMonitor` builder, e.g. `indexer.monitor( DefaultMassIndexingMonitor.builder().countOnStart( false ).build() ) )` Implementations of `MassIndexingMonitor` must be thread-safe.

Setter	Default value	Description
`failureHandler(MassIndexingFailureHandler)`	A failure handler.	The component responsible for handling failures occurring during mass indexing. A `MassIndexer` performs multiple operations in parallel, some of which can fail without stopping the whole mass indexing process. As a result, it may be necessary to trace individual failures. The default, built-in failure handler just forwards the failures to the global background failure handler, which by default will log them at the `ERROR` level, but a custom handler can be set by implementing the `MassIndexingFailureHandler` interface and passing an instance using the `failureHandler` method. This can be used to simply log failures in a context specific to the mass indexer, e.g. a web interface in a maintenance console from which mass indexing was requested, or for more advanced use cases, such as cancelling mass indexing on the first failure. Implementations of `MassIndexingFailureHandler` must be thread-safe.
`environment(MassIndexingEnvironment)`	An empty environment (no threadlocals, …).	**This feature is *incubating*: it is still under active development.** The contract of incubating elements (e.g. types, methods, configuration properties, etc.) may be altered in a backward-incompatible way — or even removed — in subsequent releases. The component responsible for setting up an environment (threadlocals, …) on mass indexing threads before mass indexing starts, and tearing down that environment after mass indexing. Implementations should handle their exceptions unless it is an unrecoverable situation in which further mass indexing does not make sense: any exception thrown by the `MassIndexingEnvironment` will abort mass indexing.
`failureFloodingThreshold(long)`	`100` with the default failure handler (see description)	**This feature is *incubating*: it is still under active development.** The maximum number of failures to be handled per indexed type. Any failures exceeding this number will be ignored and not sent for processing by `MassIndexingFailureHandler`. Can be set to `Long.MAX_VALUE` if none of the failures should be ignored. Defaults to a threshold defined by the failure handler in use; see `MassIndexingFailureHandler#failureFloodingThreshold`, `FailureHandler#failureFloodingThreshold`. For the default log-based failure handler, the default threshold is 100.

Setter	Default value	Description
`failFast(boolean)`	`false`	**This feature is *incubating*: it is still under active development.** An option to stop the indexing right after an error is encountered during the process, without waiting for the process to attempt indexing the remaining entities. With fail-fast enabled, the mass indexer will attempt to cancel any mass-indexing internal processes after the first error reported to the `MassIndexingFailureHandler`.

14.4.7. Tuning the `MassIndexer` for best performance

Basics

The `MassIndexer` was designed to finish the re-indexing task as quickly as possible, but there is no one-size-fits-all solution, so some configuration is required to get the best of it.

Performance optimization can get quite complex, so keep the following in mind while you attempt to configure the `MassIndexer`:

- Always test your changes to assess their actual effect: advice provided in this section is true in general, but each application and environment is different, and some options, when combined, may produce unexpected results.
- Take baby steps: before tuning mass indexing with 40 indexed entity types with two million instances each, try a more reasonable scenario with only one entity type, optionally limiting the number of entities to index to assess performance more quickly.
- Tune your entity types individually **before** you try to tune a mass indexing operation that indexes multiple entity types in parallel.

Threads and connections

Increasing parallelism usually helps as the bottleneck usually is the latency to the database/datastore connection: it's probably worth it to experiment with a number of threads significantly higher than the number of actual cores available.

However, each thread requires one connection (e.g. a JDBC connection), and connections are usually in limited supply. In order to increase the number of threads safely:

1. You should make sure your database/datastore can actually handle the resulting number of connections.
2. Your connection pool should be configured to provide a sufficient number of connections.
3. The above should take into account the rest of your application (request threads in a web application): ignoring this may bring other processes to a halt while the `MassIndexer` is working.

There is a simple formula to understand how the different options applied to the `MassIndexer` affect the number of used worker threads and connections:

```
if ( using the default 'none' coordination strategy ) {
    threadsToCoordinate = 0;
}
else {
    threadsToCoordinate = 1;
}
threadsToLoadIdentifiers = 1;
threads = threadsToCoordinate + typesToIndexInParallel * (threadsToLoadObjects + threadsToLoadIdentifiers);
required connections = threads;
```

Here are a few suggestions for a roughly sane tuning starting point for the parameters that affect parallelism:

`typesToIndexInParallel`

Should probably be a low value, like 1 or 2, depending on how much of your CPUs have spare cycles and how slow a database round trip will be.

`threadsToLoadObjects`

Higher increases the preloading rate for the picked entities from the database, but also increases memory usage and the pressure on the threads working on subsequent indexing. Note that each thread will extract data from the entity to reindex, which depending on your mapping might require accessing lazy associations and load associated entities, thus making blocking calls to the database/datastore, so you will probably need a high number of threads working in parallel.

All internal thread groups have meaningful names prefixed with "Hibernate Search", so they should be easily identified

with most diagnostic tools, including simply thread dumps.

14.5. Indexing a large amount of data with the Jakarta Batch integration

14.5.1. Basics

This feature is only available with the Hibernate ORM integration.

It **cannot** be used with the Standalone POJO Mapper in particular.

Hibernate Search provides a Jakarta Batch job to perform mass indexing. It covers not only the existing functionality of the mass indexer described above, but also benefits from some powerful standard features of Jakarta Batch, such as failure recovery using checkpoints, chunk oriented processing, and parallel execution. This batch job accepts different entity type(s) as input, loads the relevant entities from the database, then rebuilds the full-text index from these.

Executing this job requires a batch runtime that is not provided by Hibernate Search. You are free to choose one that fits your needs, e.g. the default batch runtime embedded in your Jakarta EE container. Hibernate Search provides full integration to the JBeret implementation (see how to configure it here). As for other implementations, they can also be used, but will require a bit more configuration on your side.

If the runtime is JBeret, you need to add the following dependency:

```
<dependency>
   <groupId>org.hibernate.search</groupId>
   <artifactId>hibernate-search-mapper-orm-jakarta-batch-jberet</artifactId>
   <version>8.4.0.CR1</version>
</dependency>
```

For any other runtime, you need to add the following dependency:

```
<dependency>
   <groupId>org.hibernate.search</groupId>
   <artifactId>hibernate-search-mapper-orm-jakarta-batch-core</artifactId>
   <version>8.4.0.CR1</version>
</dependency>
```

Here is an example of how to run a batch instance:

Example 14.13: Reindexing everything using a Jakarta Batch mass-indexing job

```
Properties jobProps = MassIndexingJob.parameters() ①
        .forEntities( Book.class, Author.class ) ②
        .build();

JobOperator jobOperator = BatchRuntime.getJobOperator(); ③
long executionId = jobOperator.start( MassIndexingJob.NAME, jobProps ); ④
```

① Start building parameters for a mass-indexing job.
② Define some parameters. In this case, the list of the entity types to be indexed.
③ Get the `JobOperator` from the framework.
④ Start the job.

14.5.2. Job Parameters

The following table contains all the job parameters you can use to customize the mass-indexing job.

Table 14.6: Job Parameters in Jakarta Batch Integration

Parameter Name / Builder Method	Default value	Description
`entityTypes` / `.forEntity(Class<?>)`, `.forEntities(Class<?>, Class<?>…)`	-	**This parameter is always required**. The entity types to index in this job execution, comma-separated.

Parameter Name / Builder Method	Default value	Description
`purgeAllOnStart` / `.purgeAllOnStart(boolean)`	True	Specify whether the existing index should be purged at the beginning of the job. This operation takes place before indexing. Only affects the indexes targeted by the `entityTypes` parameter.
`dropAndCreateSchemaOnStart` / `.dropAndCreateSchemaOnStart(boolean)`	False	Specify whether the existing schema should be dropped and created at the beginning of the job. This operation takes place before indexing. Only affects the indexes targeted by the `entityTypes` parameter.
`mergeSegmentsAfterPurge` / `.mergeSegmentsAfterPurge(boolean)`	True	Specify whether the mass indexer should merge segments at the beginning of the job. This operation takes place after the purge operation and before indexing.
`mergeSegmentsOnFinish` / `.mergeSegmentsOnFinish(boolean)`	True	Specify whether the mass indexer should merge segments at the end of the job. This operation takes place after indexing.
`cacheMode` / `.cacheMode(CacheMode)`	`IGNORE`	Specify the Hibernate `CacheMode` when loading entities. The default is `IGNORE`, and it will be the most efficient choice in most cases, but using another mode such as `GET` may be more efficient if many of the entities being indexed are already present in the Hibernate ORM second-level cache before mass indexing. Enabling caches has an effect only if the entity id is also the document id, which is the default. `PUT` or `NORMAL` values may lead to bad performance, because all the entities are also loaded into Hibernate second level cache.
`idFetchSize` / `.idFetchSize(int)`	1000	Specifies the fetch size to be used when loading primary keys. Some databases accept special values, for example MySQL might benefit from using `Integer#MIN_VALUE`, otherwise it will attempt to preload everything in memory.
`entityFetchSize` / `.entityFetchSize(int)`	200, or the value of `checkpointInterval` if it is smaller	Specifies the fetch size to be used when loading entities from database. The value defined must be greater than 0, and equal to or less than the value of `checkpointInterval`.
`customQueryHQL` / `.restrictedBy(String)`	-	Use HQL / JPQL to index entities of a target entity type. Your query should contain only one entity type. Mixing this approach with the criteria restriction is not allowed. Please notice that there's no query validation for your input. See [mapper-orm-indexing-jakarta-batch-indexing-mode] for more detail and limitations.
`maxResultsPerEntity` / `.maxResultsPerEntity(int)`	-	The maximum number of results to load per entity type. This parameter let you define a threshold value to avoid loading too many entities accidentally. The value defined must be greater than 0. The parameter is not used by default. It is equivalent to keyword `LIMIT` in SQL.
`rowsPerPartition` / `.rowsPerPartition(int)`	20,000	The maximum number of rows to process per partition. The value defined must be greater than 0, and equal to or greater than the value of `checkpointInterval`.
`maxThreads` / `.maxThreads(int)`	The number of partitions	The maximum number of threads to use for processing the job. Note the batch runtime cannot guarantee the request number of threads are available; it will use as many as it can up to the request maximum.
`checkpointInterval` / `.checkpointInterval(int)`	2,000, or the value of `rowsPerPartition` if it is smaller	The number of entities to process before triggering a checkpoint. The value defined must be greater than 0, and equal to or less than the value of `rowsPerPartition`.
`entityManagerFactoryReference` / `.entityManagerFactoryReference(String)`	-	**This parameter is required** when there is more than one persistence unit. The string that will identify the `EntityManagerFactory`.

Parameter Name / Builder Method	Default value	Description
`entityManagerFactoryNamespace` / `.entityManagerFactoryNamespace(String)`	-	See Selecting the persistence unit (EntityManagerFactory)

14.5.3. Conditional indexing

You can select a subset of target entities to be indexed by passing a condition as string to the mass indexing job. The condition will be applied when querying the database for entities to index.

The condition string is expected to follow the Hibernate Query Language (HQL) syntax. Accessible entity properties are those of the entity being reindexed (and nothing more).

Example 14.14: Conditional indexing using a `reindexOnly` HQL parameter

```
Properties jobProps = MassIndexingJob.parameters() ①
        .forEntities( Author.class ) ②
        .reindexOnly( "birthDate < :cutoff", ③
                Map.of( "cutoff", Year.of( 1950 ).atDay( 1 ) ) ) ④
        .build();

JobOperator jobOperator = BatchRuntime.getJobOperator(); ⑤
long executionId = jobOperator.start( MassIndexingJob.NAME, jobProps ); ⑥
```

① Start building parameters for a mass-indexing job.
② Define the entity type to be indexed.
③ Reindex only the authors born prior to a given local date.
④ In this example the cutoff date is passed as a query parameter.
⑤ Get `JobOperator` from the framework.
⑥ Start the job.

Even if the reindexing is applied on a subset of entities, by default **all entities** will be purged at the start. The purge can be disabled completely, but when enabled there is no way to filter the entities that will be purged.

See HSEARCH-3304 for more information.

14.5.4. Parallel indexing

For better performance, indexing is performed in parallel using multiple threads. The set of entities to index is split into multiple partitions. Each thread processes one partition at a time.

The following section will explain how to tune the parallel execution.

The "sweet spot" of number of threads, fetch size, partition size, etc. to achieve best performance is highly dependent on your overall architecture, database design and even data values.

You should experiment with these settings to find out what's best in your particular case.

Threads

The maximum number of threads used by the job execution is defined through method `maxThreads()`. Within the N threads given, there's 1 thread reserved for the core, so only N - 1 threads are available for different partitions. If N = 1, the program will work, and all batch elements will run in the same thread. The default number of threads used in Hibernate Search is 10. You can overwrite it with your preferred number.

```
MassIndexingJob.parameters()
        .maxThreads( 5 )
        ...
```

Note that the batch runtime cannot guarantee the requested number of threads are available, it will use as many as

possible up to the requested maximum (Jakarta Batch Specification v2.1 Final Release, page 29). Note also that all batch jobs share the same thread pool, so it's not always a good idea to execute jobs concurrently.

Rows per partition

Each partition consists of a fixed number of elements to index. You may tune exactly how many elements a partition will hold with `rowsPerPartition`.

```
MassIndexingJob.parameters()
        .rowsPerPartition( 5000 )
        ...
```

This property has **nothing** to do with "chunk size", which is how many elements are processed together between each write. That aspect of processing is addressed by chunking.

Instead, `rowsPerPartition` is more about how parallel your mass indexing job will be.

Please see the Chunking section to see how to tune chunking.

When `rowsPerPartition` is low, there will be many small partitions, so processing threads will be less likely to starve (stay idle because there's no more partition to process), but on the other hand you will only be able to take advantage of a small fetch size, which will increase the number of database accesses. Also, due to the failure recovery mechanisms, there is some overhead in starting a new partition, so with an unnecessarily large number of partitions, this overhead will add up.

When `rowsPerPartition` is high, there will be a few big partitions, so you will be able to take advantage of a higher chunk size, and thus a higher fetch size, which will reduce the number of database accesses, and the overhead of starting a new partition will be less noticeable, but on the other hand you may not use all the threads available.

Each partition deals with one root entity type, so two different entity types will never run under the same partition.

14.5.5. Chunking and session clearing

The mass indexing job supports restart a suspended or failed job more or less from where it stopped.

This is made possible by splitting each partition in several consecutive *chunks* of entities, and saving process information in a *checkpoint* at the end of each chunk. When a job is restarted, it will resume from the last checkpoint.

The size of each chunk is determined by the `checkpointInterval` parameter.

```
MassIndexingJob.parameters()
        .checkpointInterval( 1000 )
        ...
```

But the size of a chunk is not only about saving progress, it is also about performance:

- a new Hibernate session is opened for each chunk;
- a new transaction is started for each chunk;
- inside a chunk, the session is cleared periodically according to the `entityFetchSize` parameter, which must thereby be smaller than (or equal to) the chunk size;
- documents are flushed to the index at the end of each chunk.

In general the checkpoint interval should be small compared to the number of rows per partition.

Indeed, due to the failure recovery mechanism, the elements before the first checkpoint of each partition will take longer to process than the other, so in a 1000-element partition, having a 100-element checkpoint interval will be faster than having a 1000-element checkpoint interval.

On the other hand, **chunks shouldn't be too small** in absolute terms. Performing a checkpoint means your Jakarta Batch runtime will write information about the progress of the job execution to its persistent storage, which also has a cost. Also, a new transaction and session are created for each chunk which doesn't come for free, and implies that setting the fetch size to a value higher than the chunk size is pointless. Finally, the index flush performed at the end of each chunk is

> an expensive operation that involves a global lock, which essentially means that the less you do it, the faster indexing will be. Thus having a 1-element checkpoint interval is definitely not a good idea.

14.5.6. Selecting the persistence unit (EntityManagerFactory)

> Regardless of how the entity manager factory is retrieved, you must make sure that the entity manager factory used by the mass indexer will stay open during the whole mass indexing process.

JBeret

If your Jakarta Batch runtime is JBeret (used in WildFly in particular), you can use CDI to retrieve the `EntityManagerFactory`.

If you use only one persistence unit, the mass indexer will be able to access your database automatically without any special configuration.

If you want to use multiple persistence units, you will have to register the `EntityManagerFactories` as beans in the CDI context. Note that entity manager factories will probably not be considered as beans by default, in which case you will have to register them yourself. You may use an application-scoped bean to do so:

```
@ApplicationScoped
public class EntityManagerFactoriesProducer {

    @PersistenceUnit(unitName = "db1")
    private EntityManagerFactory db1Factory;

    @PersistenceUnit(unitName = "db2")
    private EntityManagerFactory db2Factory;

    @Produces
    @Singleton
    @Named("db1") // The name to use when referencing the bean
    public EntityManagerFactory createEntityManagerFactoryForDb1() {
        return db1Factory;
    }

    @Produces
    @Singleton
    @Named("db2") // The name to use when referencing the bean
    public EntityManagerFactory createEntityManagerFactoryForDb2() {
        return db2Factory;
    }
}
```

Once the entity manager factories are registered in the CDI context, you can instruct the mass indexer to use one in particular by naming it using the `entityManagerReference` parameter.

> Due to limitations of the CDI APIs, it is not currently possible to reference an entity manager factory by its persistence unit name when using the mass indexer with CDI.

Other DI-enabled Jakarta Batch implementations

If you want to use a different Jakarta Batch implementation that happens to allow dependency injection:

1. You must map the following two scope annotations to the relevant scope in the dependency injection mechanism:
   - `org.hibernate.search.jakarta.batch.core.inject.scope.spi.HibernateSearchJobScoped`
   - `org.hibernate.search.jakarta.batch.core.inject.scope.spi.HibernateSearchPartitionScoped`
2. You must make sure that the dependency injection mechanism will register all injection-annotated classes (`@Named`, …) from the `hibernate-search-mapper-orm-jakarta-batch-core` module in the dependency injection context. For instance this can be achieved in Spring DI using the `@ComponentScan` annotation.
3. You must register a single bean in the dependency injection context that will implement the `EntityManagerFactoryRegistry` interface.

Plain Java environment (no dependency injection at all)

The following will work only if your Jakarta Batch runtime does not support dependency injection at all, i.e. it ignores `@Inject` annotations in batch artifacts. This is the case for JBatch in Java SE mode, for instance.

If you use only one persistence unit, the mass indexer will be able to access your database automatically without any special configuration: you only have to make sure to create the `EntityManagerFactory` (or `SessionFactory`) in your application before launching the mass indexer.

If you want to use multiple persistence units, you will have to add two parameters when launching the mass indexer:

- `entityManagerFactoryReference`: this is the string that will identify the `EntityManagerFactory`.
- `entityManagerFactoryNamespace`: this allows to select how you want to reference the `EntityManagerFactory`. Possible values are:
  - `persistence-unit-name` (the default): use the persistence unit name defined in `persistence.xml`.
  - `session-factory-name`: use the session factory name defined in the Hibernate configuration by the `hibernate.session_factory_name` configuration property.

If you set the `hibernate.session_factory_name` property in the Hibernate configuration, and you don't use JNDI, you will also have to set `hibernate.session_factory_name_is_jndi` to `false`.

14.6. Explicit indexing

14.6.1. Basics

While listener-triggered indexing and the `MassIndexer` or the mass indexing job should take care of most needs, it is sometimes necessary to control indexing manually.

The need arises in particular when listener-triggered indexing is disabled or simply not supported (e.g. with the Standalone POJO Mapper), or when listener-triggered cannot detect entity changes — such as JPQL/SQL `insert`, `update` or `delete` queries.

To address these use cases, Hibernate Search exposes several APIs explained if the following sections.

14.6.2. Configuration

As explicit indexing uses indexing plans under the hood, several configuration options affecting indexing plans will affect explicit indexing as well:

- The indexing plan synchronization strategy.
- The indexing plan filter.

14.6.3. Using a `SearchIndexingPlan` manually

Explicit access to the indexing plan is done in the context of a `SearchSession` using the `SearchIndexingPlan` interface. This interface represents the (mutable) set of changes that are planned in the context of a session, and will be applied to indexes upon transaction commit (for the Hibernate ORM integration) or upon closing the `SearchSession` (for the Standalone POJO Mapper).

Here is how explicit indexing based on an indexing plan works at a high level:

1. When the application wants an index change, it calls one of the `add`/`addOrUpdate`/`delete` methods on the indexing plan of the current `SearchSession`.

   For the Hibernate ORM integration the current `SearchSession` is bound to the Hibernate ORM `Session`, while for the Standalone POJO Mapper the `SearchSession` is is created explicitly by the application.

2. Eventually, the application decides changes are complete, and the plan processes change events added so far, either inferring which entities need to be reindexed and building the corresponding documents (no coordination) or building events to be sent to the outbox (`outbox-polling` coordination).

   The application may trigger this explicitly using the indexing plan's `process` method, but it is generally not necessary as it happens automatically: for the Hibernate ORM integration this happens when the Hibernate ORM `Session` gets flushed (explicitly or as part of a transaction commit), while for the Standalone POJO Mapper this happens when the `SearchSession` is closed.

3. Finally the plan gets executed, triggering indexing, potentially asynchronously.

   The application may trigger this explicitly using the indexing plan's `execute` method, but it is generally not necessary as it happens automatically: for the Hibernate ORM integration this happens on transaction commit, while for the Standalone POJO Mapper this happens when the `SearchSession` is closed.

The `SearchIndexingPlan` interface offers the following methods:

`add(Object entity)`

(Available with the Standalone POJO Mapper only.)

Add a document to the index if the entity type is mapped to an index (`@Indexed`).

This may create duplicates in the index if the document already exists. Prefer `addOrUpdate` unless you are really sure of yourself and need a (slight) performance boost.

`addOrUpdate(Object entity)`

Add or update a document in the index if the entity type is mapped to an index (`@Indexed`), and re-index documents that embed this entity (through `@IndexedEmbedded` for example).

`delete(Object entity)`

Delete a document from the index if the entity type is mapped to an index (`@Indexed`), and re-index documents that embed this entity (through `@IndexedEmbedded` for example).

`purge(Class<?> entityType, Object id)`

Delete the entity from the index, but do not try to re-index documents that embed this entity.

Compared to `delete`, this is mainly useful if the entity has already been deleted from the database and is not available, even in a detached state, in the session. In that case, reindexing associated entities will be the user's responsibility, since Hibernate Search cannot know which entities are associated to an entity that no longer exists.

`purge(String entityName, Object id)`

Same as `purge(Class<?> entityType, Object id)`, but the entity type is referenced by its name (see `@javax.persistence.Entity#name`).

`process()``

(Available with the Hibernate ORM integration only.)

Process change events added so far, either inferring which entities need to be reindexed and building the corresponding documents (no coordination) or building events to be sent to the outbox (`outbox-polling` coordination).

This method is generally executed automatically (see the high-level description near top of this section), so calling it explicitly is only useful for batching when processing a large number of items, as explained in Hibernate ORM and the periodic "flush-clear" pattern with `SearchIndexingPlan`.

`execute()`

(Available with the Hibernate ORM integration only.)

Execute the indexing plan, triggering indexing, potentially asynchronously.

This method is generally executed automatically (see the high-level description near top of this section), so calling it explicitly is only useful in very rare cases, for batching when processing a large number of items **and transactions are not an option**, as explained in Hibernate ORM and the periodic "flush-clear" pattern with `SearchIndexingPlan`.

Below are examples of using `addOrUpdate` and `delete`.

Example 14.19: Explicitly adding or updating an entity in the index using `SearchIndexingPlan`

```
// Not shown: open a transaction if relevant

SearchSession searchSession = /* ... */ ①
SearchIndexingPlan indexingPlan = searchSession.indexingPlan(); ②

Book book = entityManager.getReference( Book.class, 5 ); ③

indexingPlan.addOrUpdate( book ); ④

// Not shown: commit the transaction or close the session if relevant
```

① Retrieve the `SearchSession`.

② Get the search session's indexing plan.

③ Fetch from the database the `Book` we want to index; this could be replaced with any other way of loading an entity when using the Standalone POJO Mapper.

④ Submit the `Book` to the indexing plan for an add-or-update operation. The operation won't be executed immediately, but will be delayed until the transaction is committed (Hibernate ORM integration) or until the `SearchSession` is closed (Standalone POJO Mapper).

Example 14.20: Explicitly deleting an entity from the index using `SearchIndexingPlan`

```
// Not shown: open a transaction if relevant

SearchSession searchSession = /* ... */ ①
SearchIndexingPlan indexingPlan = searchSession.indexingPlan(); ②

Book book = entityManager.getReference( Book.class, 5 ); ③

indexingPlan.delete( book ); ④

// Not shown: commit the transaction or close the session if relevant
```

① Retrieve the `SearchSession`.
② Get the search session's indexing plan.
③ Fetch from the database the `Book` we want to un-index; this could be replaced with any other way of loading an entity when using the Standalone POJO Mapper.
④ Submit the `Book` to the indexing plan for a delete operation. The operation won't be executed immediately, but will be delayed until the transaction is committed (Hibernate ORM integration) or until the `SearchSession` is closed (Standalone POJO Mapper).

Multiple operations can be performed in a single indexing plan. The same entity can even be changed multiple times, for example added and then removed: Hibernate Search will simplify the operation as expected.

This will work fine for any reasonable number of entities, but changing or simply loading large numbers of entities in a single session requires special care with Hibernate ORM, and then some extra care with Hibernate Search. See Hibernate ORM and the periodic "flush-clear" pattern with `SearchIndexingPlan` for more information.

14.6.4. Hibernate ORM and the periodic "flush-clear" pattern with `SearchIndexingPlan`

This feature is only available with the Hibernate ORM integration.

It **cannot** be used with the Standalone POJO Mapper in particular.

A fairly common use case when manipulating large datasets with JPA is the periodic "flush-clear" pattern, where a loop reads or writes entities for every iteration and flushes then clears the session every `n` iterations. This pattern allows processing a large number of entities while keeping the memory footprint reasonably low.

Below is an example of this pattern to persist a large number of entities when not using Hibernate Search.

Example 14.21: A batch process with JPA

```
entityManager.getTransaction().begin();
try {
    for ( int i = 0; i < NUMBER_OF_BOOKS; ++i ) { ①
        Book book = newBook( i );
        entityManager.persist( book ); ②

        if ( ( i + 1 ) % BATCH_SIZE == 0 ) {
            entityManager.flush(); ③
            entityManager.clear(); ④
        }
    }
    entityManager.getTransaction().commit();
}
catch (RuntimeException e) {
    entityManager.getTransaction().rollback();
    throw e;
}
```

① Execute a loop for a large number of elements, inside a transaction.
② For every iteration of the loop, instantiate a new entity and persist it.
③ Every `BATCH_SIZE` iterations of the loop, `flush` the entity manager to send the changes to the database-side buffer.
④ After a `flush`, `clear` the ORM session to release some memory.

With Hibernate Search 6 (on contrary to Hibernate Search 5 and earlier), this pattern will work as expected:

- with coordination disabled (the default), documents will be built on flushes, and sent to the index upon transaction commit.
- with outbox-polling coordination, entity change events will be persisted on flushes, and committed along with the rest of the changes upon transaction commit.

However, each `flush` call will potentially add data to an internal buffer, which for large volumes of data may lead to an `OutOfMemoryException`, depending on the JVM heap size, the coordination strategy and the complexity and number of documents.

If you run into memory issues, the first solution is to break down the batch process into multiple transactions, each handling a smaller number of elements: the internal document buffer will be cleared after each transaction.

See below for an example.

With this pattern, if one transaction fails, part of the data will already be in the database and in indexes, with no way to roll back the changes.

However, the indexes will be consistent with the database, and it will be possible to (manually) restart the process from the last transaction that failed.

Example 14.22: A batch process with Hibernate Search using multiple transactions

```
try {
    int i = 0;
    while ( i < NUMBER_OF_BOOKS ) { ①
        entityManager.getTransaction().begin(); ②
        int end = Math.min( i + BATCH_SIZE, NUMBER_OF_BOOKS ); ③
        for ( ; i < end; ++i ) {
            Book book = newBook( i );
            entityManager.persist( book ); ④
        }
        entityManager.getTransaction().commit(); ⑤
    }
}
catch (RuntimeException e) {
    entityManager.getTransaction().rollback();
    throw e;
}
```

① Add an outer loop that creates one transaction per iteration.
② Begin the transaction at the beginning of each iteration of the outer loop.
③ Only handle a limited number of elements per transaction.
④ For every iteration of the loop, instantiate a new entity and persist it. Note we're relying on listener-triggered indexing to index the entity, but this would work just as well if listener-triggered indexing was disabled, only requiring an extra call to index the entity. See Indexing plans.
⑤ Commit the transaction at the end of each iteration of the outer loop. The entities will be flushed and indexed automatically.

The multi-transaction solution and the original `flush()`/`clear()` loop pattern can be combined, breaking down the process in multiple medium-sized transactions, and periodically calling `flush`/`clear` inside each transaction.

This combined solution is the most flexible, hence the most suitable if you want to fine-tune your batch process.

If breaking down the batch process into multiple transactions is not an option, a second solution is to just write to indexes after the call to `session.flush()`/`session.clear()`, without waiting for the database transaction to be committed: the internal document buffer will be cleared after each write to indexes.

This is done by calling the `execute()` method on the indexing plan, as shown in the example below.

With this pattern, if an exception is thrown, part of the data will already be in the index, with no way to roll back the

changes, while the database changes will have been rolled back. The index will thus be inconsistent with the database.

To recover from that situation, you will have to either execute the exact same database changes that failed manually (to get the database back in sync with the index), or reindex the entities affected by the transaction manually (to get the index back in sync with the database).

Of course, if you can afford to take the indexes offline for a longer period of time, a simpler solution would be to wipe the indexes clean and reindex everything.

Example 14.23: A batch process with Hibernate Search using `execute()`

```
SearchSession searchSession = Search.session( entityManager ); ①
SearchIndexingPlan indexingPlan = searchSession.indexingPlan(); ②

entityManager.getTransaction().begin();
try {
    for ( int i = 0; i < NUMBER_OF_BOOKS; ++i ) {
        Book book = newBook( i );
        entityManager.persist( book ); ③

        if ( ( i + 1 ) % BATCH_SIZE == 0 ) {
            entityManager.flush();
            entityManager.clear();
            indexingPlan.execute(); ④
        }
    }
    entityManager.getTransaction().commit(); ⑤
}
catch (RuntimeException e) {
    entityManager.getTransaction().rollback();
    throw e;
}
```

① Get the `SearchSession`.

② Get the search session's indexing plan.

③ For every iteration of the loop, instantiate a new entity and persist it. Note we're relying on listener-triggered indexing to index the entity, but this would work just as well if listener-triggered indexing was disabled, only requiring an extra call to index the entity. See Indexing plans.

④ After a `flush()`/`clear()`, call `indexingPlan.execute()`. The entities will be processed and **the changes will be sent to the indexes immediately**. Hibernate Search will wait for index changes to be "completed" as required by the configured synchronization strategy.

⑤ After the loop, commit the transaction. The remaining entities that were not flushed/cleared will be flushed and indexed automatically.

Chapter 15. Searching

Beyond simply indexing, Hibernate Search also exposes high-level APIs to search these indexes without having to resort to native APIs.

One key feature of these search APIs is the ability to use indexes to perform the search, but to return entities loaded **from the database**, effectively offering a new type of query for Hibernate ORM entities.

15.1. Query DSL

15.1.1. Basics

Preparing and executing a query requires just a few lines:

Example 15.1: Executing a search query

```
// Not shown: open a transaction if relevant
SearchSession searchSession = /* ... */ ①

SearchResult<Book> result = searchSession.search( Book.class ) ②
        .where( f -> f.match() ③
                .field( "title" )
                .matching( "robot" ) )
        .fetch( 20 ); ④

long totalHitCount = result.total().hitCount(); ⑤
List<Book> hits = result.hits(); ⑥
// Not shown: commit the transaction if relevant
```

① Retrieve the `SearchSession`.
② Initiate a search query on the index mapped to the `Book` entity.
③ Define that only documents matching the given predicate should be returned. The predicate is created using a factory `f` passed as an argument to the lambda expression. See Predicate DSL for more information about predicates.
④ Build the query and fetch the results, limiting to the top 20 hits.
⑤ Retrieve the total number of matching entities. See Fetching the total (hit count, …) for ways to optimize computation of the total hit count.
⑥ Retrieve matching entities.

This will work fine with the Hibernate ORM integration: by default, the hits of a search query will be entities managed by Hibernate ORM, bound to the entity manager used to create the search session. This provides all the benefits of Hibernate ORM, in particular the ability to navigate the entity graph to retrieve associated entities if necessary.

For the Standalone POJO Mapper, the snippet above will fail by default.

You will need to either:

- configure target entity types to enable loading, if you want to load entities from an external datasource.
- add a projection constructor to target entity types, if you want to reconstruct entities from the content of the index.
- use explicit projections to retrieve specific data from the index instead.

The query DSL offers many features, detailed in the following sections. Some commonly used features include:

- predicates, the main component of a search query, i.e. the condition that every document must satisfy in order to be included in search results.
- fetching the results differently: getting the hits directly as a list, using pagination, scrolling, etc.
- sorts, to order the hits in various ways: by score, by the value of a field, by distance to a point, etc.
- projections, to retrieve hits that are not just managed entities: data can be extracted from the index (field values), or even from both the index and the database.
- aggregations, to group hits and compute aggregated metrics for each group — hit count by category, for example.

15.1.2. Advanced entity types targeting

Targeting multiple entity types

When multiple entity types have similar indexed fields, it is possible to search across these multiple types in a single search query: the search result will contain hits from any of the targeted types.

Example 15.2: Targeting multiple entity types in a single search query

```
SearchResult<Person> result = searchSession.search( Arrays.asList( ①
        Manager.class, Associate.class
) )
        .where( f -> f.match() ②
                .field( "name" )
                .matching( "james" ) )
        .fetch( 20 ); ③
```

① Initiate a search query targeting the indexes mapped to the `Manager` and `Associate` entity types. Since both entity types implement the `Person` interface, search hits will be instances of `Person`.
② Continue building the query as usual. There are restrictions regarding the fields that can be used: see the note below.
③ Fetch the search result. Hits will all be instances of `Person`.

Multi-entity (multi-index) searches will only work well as long as the fields referenced in predicates/sorts/etc. are identical in all targeted indexes (same type, same analyzer, …). Fields that are defined in only one of the targeted indexes will also work correctly.

If you want to reference index fields that are even **slightly** different in one of the targeted indexes (different type, different analyzer, …), see Targeting multiple fields.

Targeting entity types by name

Though rarely necessary, it is also possible to use entity names instead of classes to designate the entity types targeted by the search:

Example 15.3: Targeting entity types by name

```
SearchResult<Person> result = searchSession.search( ①
        searchSession.scope( ②
                Person.class,
                Arrays.asList( "Manager", "Associate" )
        )
)
        .where( f -> f.match() ③
                .field( "name" )
                .matching( "james" ) )
        .fetch( 20 ); ④
```

① Initiate a search query.
② Pass a custom scope encompassing the indexes mapped to the `Manager` and `Associate` entity types, expecting those entity types to implement the `Person` interface (Hibernate Search will check that).
③ Continue building the query as usual.
④ Fetch the search result. Hits will all be instances of `Person`.

15.1.3. Fetching results

Basics

In Hibernate Search, the default search result is a bit more complicated than just "a list of hits". This is why the default methods return a composite `SearchResult` object offering getters to retrieve the part of the result you want, as shown in the example below.

Example 15.4: Getting information from a `SearchResult`

```
SearchResult<Book> result = searchSession.search( Book.class ) ①
        .where( f -> f.matchAll() )
        .fetch( 20 ); ②
```

```
long totalHitCount = result.total().hitCount(); ③
List<Book> hits = result.hits(); ④
// ... ⑤
```

① Start building the query as usual.
② Fetch the results, limiting to the top 20 hits.
③ Retrieve the total hit count, i.e. the total number of matching entities/documents, which could be 10,000 even if you only retrieved the top 20 hits. This is useful to give end users and idea of how many more hits they query produced. See Fetching the total (hit count, …) for ways to optimize computation of the total hit count.
④ Retrieve the top hits, in this case the top 20 matching entities/documents.
⑤ Other kinds of results and information can be retrieved from `SearchResult`. They are explained in dedicated sections, such as Aggregation DSL.

It is possible to retrieve the total hit count alone, for cases where only the number of hits is of interest, not the hits themselves:

Example 15.5: Getting the total hit count directly

```
long totalHitCount = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .fetchTotalHitCount();
```

The top hits can also be obtained directly, without going through a `SearchResult`, which can be handy if only the top hits are useful, and not the total hit count:

Example 15.6: Getting the top hits directly

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .fetchHits( 20 );
```

If only zero to one hit is expected, it is possible to retrieve it as an `Optional`. An exception will be thrown if more than one hits are returned.

Example 15.7: Getting the only hit directly

```
Optional<Book> hit = searchSession.search( Book.class )
        .where( f -> f.id().matching( 1 ) )
        .fetchSingleHit();
```

Fetching all hits

Fetching all hits is rarely a good idea: if the query matches many entities/documents, this may lead to loading millions of entities in memory, which will likely crash the JVM, or at the very least slow it down to a crawl.

If you know your query will always have less than N hits, consider setting the limit to N to avoid memory issues.

If there is no bound to the number of hits you expect, you should consider Pagination or Scrolling to retrieve data in batches.

If you still want to fetch all hits in one call, be aware that the Elasticsearch backend will only ever return 10,000 hits at a time, due to internal safety mechanisms in the Elasticsearch cluster.

Example 15.8: Getting all hits in a `SearchResult`

```
SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.id().matchingAny( Arrays.asList( 1, 2 ) ) )
        .fetchAll();

long totalHitCount = result.total().hitCount();
```

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

Example 15.9: Getting all hits directly

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.id().matchingAny( Arrays.asList( 1, 2 ) ) )
        .fetchAllHits();
```

Fetching the total (hit count, …)

A `SearchResultTotal` contains the count of the **total** hits have been matched the query, either belonging to the current page or not. For pagination see Pagination.

The total hit count is exact by default, but can be replaced with a lower-bound estimate in the following cases:

- The `totalHitCountThreshold` option is enabled. See `totalHitCountThreshold(…)`: optimizing total hit count computation.
- The `truncateAfter` option is enabled and a timeout occurs.

Example 15.10: Working with the result total

```
SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .fetch( 20 );

SearchResultTotal resultTotal = result.total(); ①
long totalHitCount = resultTotal.hitCount(); ②
long totalHitCountLowerBound = resultTotal.hitCountLowerBound(); ③
boolean hitCountExact = resultTotal.isHitCountExact(); ④
boolean hitCountLowerBound = resultTotal.isHitCountLowerBound(); ⑤
```

① Extract the `SearchResultTotal` from the `SearchResult`.
② Retrieve the exact total hit count. This call will raise an exception if the only available hit count is a lower-bound estimate.
③ Retrieve a lower-bound estimate of the total hit count. This will return the exact hit count if available.
④ Test if the count is exact.
⑤ Test if the count is a lower bound.

`totalHitCountThreshold(…)`: optimizing total hit count computation

When working with large result sets, counting the number of hits exactly can be very resource-consuming.

When sorting by score (the default) and retrieving the result through `fetch(…)`, it is possible to yield significant performance improvements by allowing Hibernate Search to return a lower-bound estimate of the total hit count, instead of the exact total hit count. In that case, the underlying engine (Lucene or Elasticsearch) will be able to skip large chunks of non-competitive hits, leading to fewer index scans and thus better performance.

To enable this performance optimization, call `totalHitCountThreshold(…)` when building the query, as shown in the example below.

This optimization has no effect in the following cases:

- when calling `fetchHits(…)`: it is already optimized by default.
- when calling `fetchTotalHitCount()`: it always returns an exact hit count.
- when calling `scroll(…)` with the Elasticsearch backend: Elasticsearch does not support this optimization when scrolling. The optimization is enabled for `scroll(…)` calls with the Lucene backend, however.

Example 15.11: Defining a total hit count threshold

```
SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .totalHitCountThreshold( 1000 ) ①
        .fetch( 20 );
```

```
SearchResultTotal resultTotal = result.total(); ②
long totalHitCountLowerBound = resultTotal.hitCountLowerBound(); ③
boolean hitCountExact = resultTotal.isHitCountExact(); ④
boolean hitCountLowerBound = resultTotal.isHitCountLowerBound(); ⑤
```

① Define a `totalHitCountThreshold` for the current query
② Extract the `SearchResultTotal` from the `SearchResult`.
③ Retrieve a lower-bound estimate of the total hit count. This will return the exact hit count if available.
④ Test if the count is exact.
⑤ Test if the count is a lower-bound estimate.

Pagination

Pagination is the concept of splitting hits in successive "pages", all pages containing a fixed number of elements (except potentially the last one). When displaying results on a web page, the user will be able to go to an arbitrary page and see the corresponding results, for example "results 151 to 170 of 14,265".

Pagination is achieved in Hibernate Search by passing an offset and a limit to the `fetch` or `fetchHits` method:

- The offset defines the number of documents that should be skipped because they were displayed in previous pages. It is a **number of documents**, not a number of pages, so you will usually want to compute it from the page number and page size this way: `offset = zero-based-page-number * page-size`.
- The limit defines the maximum number of hits to return, i.e. the page size.

Example 15.12: Pagination retrieving a `SearchResult`

```
SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .fetch( 40, 20 ); ①
```

① Set the offset to `40` and the limit to `20`.

Example 15.13: Pagination retrieving hits directly

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .fetchHits( 40, 20 ); ①
```

① Set the offset to `40` and the limit to `20`.

The index may be modified between the retrieval of two pages. As a result of that modification, it is possible that some hits change position, and end up being present on two subsequent pages.

If you're running a batch process and want to avoid this, use Scrolling.

Scrolling

Scrolling is the concept of keeping a cursor on the search query at the lowest level, and advancing that cursor progressively to collect subsequent "chunks" of search hits.

Scrolling relies on the internal state of the cursor (which must be closed at some point), and thus is not appropriate for stateless operations such as displaying a page of results to a user in a webpage. However, thanks to this internal state, scrolling is able to guarantee that all returned hits are consistent: there is absolutely no way for a given hit to appear twice.

Scrolling is therefore most useful when processing a large result set as small chunks.

Below is an example of using scrolling in Hibernate Search.

`SearchScroll` exposes a `close()` method that **must** be called to avoid resource leaks.

With the Elasticsearch backend, scrolls can time out and become unusable after some time; See here for more

information.

Example 15.14: Scrolling to retrieve search results in small chunks

```
try ( SearchScroll<Book> scroll = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .scroll( 20 ) ) { ①
    for ( SearchScrollResult<Book> chunk = scroll.next(); ②
            chunk.hasHits(); chunk = scroll.next() ) { ③
        for ( Book hit : chunk.hits() ) { ④
            // ... do something with the hits ...
        }

        totalHitCount = chunk.total().hitCount(); ⑤

        entityManager.flush(); ⑥
        entityManager.clear(); ⑥
    }
}
```

① Start a scroll that will return chunks of `20` hits. Note the scroll is used in a `try-with-resource` block to avoid resource leaks.
② Retrieve the first chunk by calling `next()`. Each chunk will include at most 20 hits, since that was the selected chunk size.
③ Detect the end of the scroll by calling `hasHits()` on the last retrieved chunk, and retrieve the next chunk by calling `next()` again on the scroll.
④ Retrieve the hits of a chunk.
⑤ Optionally, retrieve the total number of matching entities.
⑥ Optionally, if using Hibernate ORM and retrieving entities, you might want to use the periodic "flush-clear" pattern to ensure entities don't stay in the session taking more and more memory.

15.1.4. Routing

For a preliminary introduction to sharding, including how it works in Hibernate Search and what its limitations are, see Sharding and routing.

If, for a given index, there is one immutable value that documents are often filtered on, for example a "category" or a "user id", it is possible to match documents with this value using a routing key instead of a predicate.

The main advantage of a routing key over a predicate is that, on top of filtering documents, the routing key will also filter shards. If sharding is enabled, this means only part of the index will be scanned during query execution, potentially increasing search performance.

A pre-requisite to using routing in search queries is to map your entity in such a way that it is assigned a routing key at indexing time.

Specifying routing keys is done by calling the `.routing(String)` or `.routing(Collection<String>)` methods when building the query:

Example 15.15: Routing a query to a subset of all shards

```
SearchResult<Book> result = searchSession.search( Book.class ) ①
        .where( f -> f.match()
                .field( "genre" )
                .matching( Genre.SCIENCE_FICTION ) ) ②
        .routing( Genre.SCIENCE_FICTION.name() ) ③
        .fetch( 20 ); ④
```

① Start building the query.
② Define that only documents matching the given `genre` should be returned.
③ In this case, the entity is mapped in such a way that the `genre` is also used as a routing key. We know all documents will have the given `genre` value, so we can specify the routing key to limit the query to relevant shards.
④ Build the query and fetch the results.

15.1.5. Entity loading options for Hibernate ORM

When using the Hibernate ORM mapper, Hibernate Search executes database queries to load entities that are returned as part of the hits of a search query.

This section presents all available options related to entity loading in search queries.

Cache lookup strategy

This feature is only available with the Hibernate ORM integration.

It **cannot** be used with the Standalone POJO Mapper in particular.

By default, Hibernate Search will load entities from the database directly, without looking at any cache. This is a good strategy when the size of caches (Hibernate ORM session or second level cache) is much lower than the total number of indexed entities.

If a significant portion of your entities are present in the second level cache, you can force Hibernate Search to retrieve entities from the persistence context (the session) and/or the second level cache if possible. Hibernate Search will still need to execute a database query to retrieve entities missing from the cache, but the query will likely have to fetch fewer entities, leading to better performance and lower stress on your database.

This is done through the cache lookup strategy, which can be configured by setting the configuration property `hibernate.search.query.loading.cache_lookup.strategy`:

- `skip` (the default) will not perform any cache lookup.
- `persistence-context` will only look into the persistence context, i.e. will check if the entities are already loaded in the session. Useful if most search hits are expected to already be loaded in session, which is generally unlikely.
- `persistence-context-then-second-level-cache` will first look into the persistence context, then into the second level cache, if enabled in Hibernate ORM for the searched entity. Useful if most search hits are expected to be cached, which may be likely if you have a small number of entities and a large cache.

Before a second-level cache can be used for a given entity type, some configuration is required in Hibernate ORM.

See the caching section of the Hibernate ORM documentation for more information.

It is also possible to override the configured strategy on a per-query basis, as shown below.

Example 15.16: Overriding the cache lookup strategy in a single search query

```
SearchResult<Book> result = searchSession.search( Book.class ) ①
        .where( f -> f.match()
                .field( "title" )
                .matching( "robot" ) )
        .loading( o -> o.cacheLookupStrategy( ②
                EntityLoadingCacheLookupStrategy.PERSISTENCE_CONTEXT_THEN_SECOND_LEVEL_CACHE
        ) )
        .fetch( 20 ); ③
```

① Start building the query.
② Access the loading options of the query, then mention that the persistence context and second level cache should be checked before entities are loaded from the database.
③ Fetch the results. The more entities found in the persistence context or second level cache, the fewer entities will be loaded from the database.

Fetch size

This feature is only available with the Hibernate ORM integration.

It **cannot** be used with the Standalone POJO Mapper in particular.

By default, Hibernate Search will use a fetch size of `100`, meaning that for a single `fetch*()` call on a single query, it will run a first query to load the first 100 entities, then if there are more hits it will run a second query to load the next 100, etc.

The fetch size can be configured by setting the configuration property `hibernate.search.query.loading.fetch_size`. This property expects a strictly positive Integer value.

It is also possible to override the configured fetch size on a per-query basis, as shown below.

Example 15.17: Overriding the fetch size in a single search query

```
SearchResult<Book> result = searchSession.search( Book.class ) ①
        .where( f -> f.match()
                .field( "title" )
                .matching( "robot" ) )
        .loading( o -> o.fetchSize( 50 ) ) ②
        .fetch( 200 ); ③
```

① Start building the query.
② Access the loading options of the query, then set the fetch size to an arbitrary value (must be `1` or more).
③ Fetch the results, limiting to the top 200 hits. One query will be executed to load the hits if there are fewer hits than the given fetch size; two queries if there are more hits than the fetch size but less than twice the fetch size, etc.

Entity graph

This feature is only available with the Hibernate ORM integration.

It **cannot** be used with the Standalone POJO Mapper in particular.

By default, Hibernate Search will load associations according to the defaults of your mappings: associations marked as lazy won't be loaded, while associations marked as eager will be loaded before returning the entities.

It is possible to force the loading of a lazy association, or to prevent the loading of an eager association, by referencing an entity graph in the query. See below for an example, and this section of the Hibernate ORM documentation for more information about entity graphs.

Example 15.18: Applying an entity graph to a search query

```
EntityManager entityManager = /* ... */

EntityGraph<Manager> graph = entityManager.createEntityGraph( Manager.class ); ①
graph.addAttributeNodes( "associates" );

SearchResult<Manager> result = Search.session( entityManager ).search( Manager.class ) ②
        .where( f -> f.match()
                .field( "name" )
                .matching( "james" ) )
        .loading( o -> o.graph( graph, GraphSemantic.FETCH ) ) ③
        .fetch( 20 ); ④
```

① Build an entity graph.
② Start building the query.
③ Access the loading options of the query, then set the entity graph to the graph built above. You must also pass a semantic: `GraphSemantic.FETCH` means only associations referenced in the graph will be loaded; `GraphSemantic.LOAD` means associations referenced in the graph **and** associations marked as `EAGER` in the mapping will be loaded.
④ Fetch the results. All managers loaded by this search query will have their `associates` association already populated.

Instead of building the entity graph on the spot, you can also define the entity graph statically using the `@NamedEntityGraph` annotation, and pass the name of your graph to Hibernate Search, as shown below. See this section of the Hibernate ORM documentation for more information about `@NamedEntityGraph`.

Example 15.19: Applying a named entity graph to a search query

```
SearchResult<Manager> result = Search.session( entityManager ).search( Manager.class ) ①
        .where( f -> f.match()
                .field( "name" )
                .matching( "james" ) )
        .loading( o -> o.graph( "preload-associates", GraphSemantic.FETCH ) ) ②
        .fetch( 20 ); ③
```

① Start building the query.

② Access the loading options of the query, then set the entity graph to "preload-associates", which was defined elsewhere using the `@NamedEntityGraph` annotation.
③ Fetch the results. All managers loaded by this search query will have their `associates` association already populated.

15.1.6. Timeout

You can limit the time it takes for a search query to execute in two ways:

- Aborting (throwing an exception) when the time limit is reached with `failAfter()`.
- Truncating the results when the time limit is reached with `truncateAfter()`.

Currently, the two approaches are incompatible: trying to set both `failAfter` and `truncateAfter` will result in unspecified behavior.

`failAfter()`: Aborting the query after a given amount of time

By calling `failAfter(…)` when building the query, it is possible to set a time limit for the query execution. Once the time limit is reached, Hibernate Search will stop the query execution and throw a `SearchTimeoutException`.

Timeouts are handled on a best-effort basis.

Depending on the resolution of the internal clock and on how often Hibernate Search is able to check that clock, it is possible that a query execution exceeds the timeout. Hibernate Search will try to minimize this excess execution time.

Example 15.20: Triggering a failure on timeout

```
try {
    SearchResult<Book> result = searchSession.search( Book.class ) ①
            .where( f -> f.match()
                    .field( "title" )
                    .matching( "robot" ) )
            .failAfter( 500, TimeUnit.MILLISECONDS ) ②
            .fetch( 20 ); ③
}
catch (SearchTimeoutException e) { ④
    // ...
}
```

① Build the query as usual.
② Call `failAfter` to set the timeout.
③ Fetch the results.
④ Catch the exception if necessary.

`explain()` does not honor this timeout: this method is used for debugging purposes and in particular to find out why a query is slow.

`truncateAfter()`: Truncating the results after a given amount of time

By calling `truncateAfter(…)` when building the query, it is possible to set a time limit for the collection of search results. Once the time limit is reached, Hibernate Search will stop collecting hits and return an incomplete result.

Timeouts are handled on a best-effort basis.

Depending on the resolution of the internal clock and on how often Hibernate Search is able to check that clock, it is possible that a query execution exceeds the timeout. Hibernate Search will try to minimize this excess execution time.

Example 15.21: Truncating the results on timeout

```
SearchResult<Book> result = searchSession.search( Book.class ) ①
        .where( f -> f.match()
```

```
                        .field( "title" )
                        .matching( "robot" ) )
                .truncateAfter( 500, TimeUnit.MILLISECONDS ) ②
                .fetch( 20 ); ③

        Duration took = result.took(); ④
        Boolean timedOut = result.timedOut(); ⑤
```

① Build the query as usual.
② Call `truncateAfter` to set the timeout.
③ Fetch the results.
④ Optionally extract *took*: how much time the query took to execute.
⑤ Optionally extract *timedOut*: whether the query timed out.

`explain()` and `fetchTotalHitCount()` do not honor this timeout. The former is used for debugging purposes and in particular to find out why a query is slow. For the latter it does not make sense to return a *partial* result.

15.1.7. Setting query parameters

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

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

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

Some query elements may leverage the use of query parameters. Call `.param(..)` at the query level to set these:

Example 15.22: Setting query parameters

```
List<Manager> managers = searchSession.search( Manager.class )
        .where(
                //...
        )
        .param( "param1", "name" )
        .param( "param2", 10 )
        .param( "param3", LocalDate.of( 2002, 02, 20 ) )
        .fetchAllHits();
```

See also:

- Creating projections with parameters
- Creating predicates with parameters
- Creating aggregations with parameters
- Creating sorts with parameters

15.1.8. Obtaining a query object

The example presented in most of this documentation fetch the query results directly at the end of the query definition DSL, not showing any "query" object that can be manipulated. This is because the query object generally only makes code more verbose without bringing anything worthwhile.

However, in some cases a query object can be useful. To get a query object, just call `toQuery()` at the end of the query definition:

Example 15.23: Getting a `SearchQuery` object

```
SearchQuery<Book> query = searchSession.search( Book.class ) ①
        .where( f -> f.matchAll() )
        .toQuery(); ②
List<Book> hits = query.fetchHits( 20 ); ③
```

① Build the query as usual.
② Retrieve a `SearchQuery` object.
③ Fetch the results.

This query object supports all `fetch*` methods supported by the query DSL. The main advantage over calling these methods directly at the end of a query definition is mostly related to troubleshooting, but the query object can also be useful if you need an adapter to another API.

Hibernate Search provides an adapter to JPA and Hibernate ORM's native APIs, i.e. a way to turn a `SearchQuery` into a `javax.persistence.TypedQuery` (JPA) or a `org.hibernate.query.Query` (native ORM API):

Example 15.24: Turning a `SearchQuery` into a JPA or Hibernate ORM query

```
SearchQuery<Book> query = searchSession.search( Book.class ) ①
        .where( f -> f.matchAll() )
        .toQuery(); ②
jakarta.persistence.TypedQuery<Book> jpaQuery = Search.toJpaQuery( query ); ③
org.hibernate.query.Query<Book> ormQuery = Search.toOrmQuery( query ); ④
```

① Build the query as usual.
② Retrieve a `SearchQuery` object.
③ Turn the `SearchQuery` object into a JPA query.
④ Turn the `SearchQuery` object into a Hibernate ORM query.

Both JPA and Hibernate ORM query adapters are deprecated and will be removed in the future version of Hibernate Search.

The resulting query **does not support all operations**, so it is recommended to only convert search queries when absolutely required, for example when integrating with code that only works with Hibernate ORM queries.

The following operations are expected to work correctly in most cases, even though they may behave slightly differently from what is expected from a JPA `TypedQuery` or Hibernate ORM `Query` in some cases (including, but not limited to, the type of thrown exceptions):

- Direct hit retrieval methods: `list`, `getResultList`, `uniqueResult`, …
- Scrolling: `scroll()`, `scroll(ScrollMode)` (but only with `ScrollMode.FORWARDS_ONLY`).
- `setFirstResult`/`setMaxResults` and getters.
- `setFetchSize`
- `unwrap`
- `setHint`

The following operations are known not to work correctly, with no plan to fix them at the moment:

- `getHints`
- Parameter-related methods: `setParameter`, …
- Result transformer: `setResultTransformer`, … Use composite projections instead.
- Lock-related methods: `setLockOptions`, …
- And more: this list is not exhaustive.

15.1.9. `explain(…)`: Explaining scores

In order to explain the score of a particular document, create a `SearchQuery` object using `toQuery()` at the end of the query definition, and then use one of the backend-specific `explain(…)` methods; the result of these methods will include a human-readable description of how the score of a specific document was computed.

Regardless of the API used, explanations are rather costly performance-wise: only use them for debugging purposes.

Example 15.25: Retrieving score explanation — Lucene

```
LuceneSearchQuery<Book> query = searchSession.search( Book.class )
        .extension( LuceneExtension.get() ) ①
```

```
        .where( f -> f.match()
                .field( "title" )
                .matching( "robot" ) )
        .toQuery(); ②

Explanation explanation1 = query.explain( 1 ); ③
Explanation explanation2 = query.explain( "Book", 1 ); ④

LuceneSearchQuery<Book> luceneQuery = query.extension( LuceneExtension.get() ); ⑤
```

① Build the query as usual, but using the Lucene extension so that the retrieved query exposes Lucene-specific operations.

② Retrieve a `SearchQuery` object.

③ Retrieve the explanation of the score of the entity with ID `1`. The explanation is of type `Explanation`, but you can convert it to a readable string using `toString()`.

④ For multi-index queries, it is necessary to refer to the entity not only by its ID, but also by the name of its type.

⑤ If you cannot change the code building the query to use the Lucene extension, you can instead use the Lucene extension on the `SearchQuery` to convert it after its creation.

Example 15.26: Retrieving score explanation — Elasticsearch

```
ElasticsearchSearchQuery<Book> query = searchSession.search( Book.class )
        .extension( ElasticsearchExtension.get() ) ①
        .where( f -> f.match()
                .field( "title" )
                .matching( "robot" ) )
        .toQuery(); ②

JsonObject explanation1 = query.explain( 1 ); ③
JsonObject explanation2 = query.explain( "Book", 1 ); ④

ElasticsearchSearchQuery<Book> elasticsearchQuery = query.extension( ElasticsearchExtension.get() ); ⑤
```

① Build the query as usual, but using the Elasticsearch extension so that the retrieved query exposes Elasticsearch-specific operations.

② Retrieve a `SearchQuery` object.

③ Retrieve the explanation of the score of the entity with ID `1`.

④ For multi-index queries, it is necessary to refer to the entity not only by its ID, but also by the name of its type.

⑤ If you cannot change the code building the query to use the Elasticsearch extension, you can instead use the Elasticsearch extension on the `SearchQuery` to convert it after its creation.

15.1.10. `took` and `timedOut`: finding out how long the query took

Example 15.27: Returning query execution time and whether a timeout occurred

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

SearchResult<Book> result = query.fetch( 20 ); ①

Duration took = result.took(); ②
Boolean timedOut = result.timedOut(); ③
```

① Fetch the results.

② Extract *took*: how much time the query took (in case of Elasticsearch, ignoring network latency between the application and the Elasticsearch cluster).

③ Extract *timedOut*: whether the query timed out (in case of Elasticsearch, ignoring network latency between the application and the Elasticsearch cluster).

15.1.11. Elasticsearch: leveraging advanced features with JSON manipulation

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

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

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

Elasticsearch ships with many features. It is possible that at some point, one feature you need will not be exposed by the Search DSL.

To work around such limitations, Hibernate Search provides ways to:

- Transform the HTTP request sent to Elasticsearch for search queries.
- Read the raw JSON of the HTTP response received from Elasticsearch for search queries.

Direct changes to the HTTP request may conflict with Hibernate Search features and be supported differently by different versions of Elasticsearch.

Similarly, the content of the HTTP response may change depending on the version of Elasticsearch, depending on which Hibernate Search features are used, and even depending on how Hibernate Search features are implemented.

Thus, features relying on direct access to HTTP requests or responses cannot be guaranteed to continue to work when upgrading Hibernate Search, even for micro upgrades (`x.y.z` to `x.y.(z+1)`).

Use this at your own risk.

Most simple use cases will only need to change the HTTP request slightly, as shown below.

Example 15.28: Transforming the Elasticsearch request manually in a search query

```
List<Book> hits = searchSession.search( Book.class )
        .extension( ElasticsearchExtension.get() ) ①
        .where( f -> f.match()
                .field( "title" )
                .matching( "robot" ) )
        .requestTransformer( context -> { ②
            Map<String, String> parameters = context.parametersMap(); ③
            parameters.put( "search_type", "dfs_query_then_fetch" );

            JsonObject body = context.body(); ④
            body.addProperty( "min_score", 0.2f );
        } )
        .fetchHits( 20 ); ⑤
```

① Build the query as usual, but using the Elasticsearch extension so that Elasticsearch-specific options are available.
② Add a request transformer to the query. Its `transform` method will be called whenever a request is about to be sent to Elasticsearch.
③ Inside the `transform` method, alter the HTTP query parameters.
④ It is also possible to alter the request's JSON body as shown here, or even the request's path (not shown in this example).
⑤ Retrieve the result as usual.

For more complicated use cases, it is possible to access the raw JSON of the HTTP response, as shown below.

Example 15.29: Accessing the Elasticsearch response body manually in a search query

```
ElasticsearchSearchResult<Book> result = searchSession.search( Book.class )
        .extension( ElasticsearchExtension.get() ) ①
        .where( f -> f.match()
                .field( "title" )
                .matching( "robt" ) )
        .requestTransformer( context -> { ②
            JsonObject body = context.body();
            body.add( "suggest", jsonObject( suggest -> { ③
                suggest.add( "my-suggest", jsonObject( mySuggest -> {
```

```
                    mySuggest.addProperty( "text", "robt" );
                    mySuggest.add( "term", jsonObject( term -> {
                        term.addProperty( "field", "title" );
                    } ) );
                } ) );
            } ) );
        } )
        .fetch( 20 ); ④

JsonObject responseBody = result.responseBody(); ⑤
JsonArray mySuggestResults = responseBody.getAsJsonObject( "suggest" ) ⑥
        .getAsJsonArray( "my-suggest" );
```

① Build the query as usual, but using the Elasticsearch extension so that Elasticsearch-specific options are available.
② Add a request transformer to the query.
③ Add content to the request body, so that Elasticsearch will return more data in the response. Here we're asking Elasticsearch to apply a suggester.
④ Retrieve the result as usual. Since we used the Elasticsearch extension when building the query, the result is an `ElasticsearchSearchResult` instead of the usual `SearchResult`.
⑤ Get the response body as a `JsonObject`.
⑥ Extract useful information from the response body. Here we're extracting the result of the suggester we configured above.

Gson's API for building JSON objects is quite verbose, so the example above relies on a small, custom helper method to make the code more readable:

```
    private static JsonObject jsonObject(Consumer<JsonObject> instructions) {
        JsonObject object = new JsonObject();
        instructions.accept( object );
        return object;
    }
```

When data needs to be extracted from each hit, it is often more convenient to use the `jsonHit` projection than parsing the whole response.

15.1.12. Lucene: retrieving low-level components

Lucene queries allow to retrieve some low-level components. This should only be useful to integrators, but is documented here for the sake of completeness.

Example 15.30: Accessing low-level components in a Lucene search query

```
LuceneSearchQuery<Book> query = searchSession.search( Book.class )
        .extension( LuceneExtension.get() ) ①
        .where( f -> f.match()
                .field( "title" )
                .matching( "robot" ) )
        .sort( f -> f.field( "title_sort" ) )
        .toQuery(); ②

Sort sort = query.luceneSort(); ③

LuceneSearchResult<Book> result = query.fetch( 20 ); ④

TopDocs topDocs = result.topDocs(); ⑤
```

① Build the query as usual, but using the Lucene extension so that Lucene-specific options are available.
② Since we used the Lucene extension when building the query, the query is a `LuceneSearchQuery` instead of the usual `SearchQuery`.
③ Retrieve the `org.apache.lucene.search.Sort` this query relies on.
④ Retrieve the result as usual. `LuceneSearchQuery` returns a `LuceneSearchResult` instead of the usual `SearchResult`.
⑤ Retrieve the `org.apache.lucene.search.TopDocs` for this result. Note that the `TopDocs` are offset according to the arguments to the `fetch` method, if any.

15.2. Predicate DSL

15.2.1. Basics

The main component of a search query is the *predicate*, i.e. the condition that every document must satisfy in order to be included in search results.

The predicate is configured when building the search query:

Example 15.31: Defining the predicate of a search query

```
SearchSession searchSession = /* ... */ ①

List<Book> result = searchSession.search( Book.class ) ②
        .where( f -> f.match().field( "title" ) ③
                .matching( "robot" ) )
        .fetchHits( 20 ); ④
```

① Retrieve the `SearchSession`.
② Start building the query.
③ Mention that the results of the query are expected to have a `title` field matching the value `robot`. If the field does not exist or cannot be searched on, an exception will be thrown.
④ Fetch the results, which will match the given predicate.

Alternatively, if you don't want to use lambdas:

Example 15.32: Defining the predicate of a search query — object-based syntax

```
SearchSession searchSession = /* ... */

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

List<Book> result = searchSession.search( scope )
        .where( scope.predicate().match().field( "title" )
                .matching( "robot" )
                .toPredicate() )
        .fetchHits( 20 );
```

The predicate DSL offers more predicate types, and multiple options for each type of predicate. To learn more about the `match` predicate, and all the other types of predicate, refer to the following sections.

15.2.2. `matchAll`: match all documents

The `matchAll` predicate simply matches all documents.

Example 15.33: Matching all documents

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .fetchHits( 20 );
```

`except(…)`: exclude documents matching a given predicate

Optionally, you can exclude a few documents from the hits:

Example 15.34: Matching all documents except those matching a given predicate

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.matchAll()
                .except( f.match().field( "title" )
                        .matching( "robot" ) )
```

```
            )
            .fetchHits( 20 );
```

Other options

- The score of a `matchAll` predicate is constant and equal to 1 by default, but can be boosted with `.boost(…)`.

15.2.3. `matchNone`: match no documents

The `matchNone` predicate is the inverse of `matchAll` and matches no documents.

Example 15.35: Matching no documents

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.matchNone() )
        .fetchHits( 20 );
```

15.2.4. `id`: match a document identifier

The `id` predicate matches documents by their identifier.

Example 15.36: Matching a document with a given identifier

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.id().matching( 1 ) )
        .fetchHits( 20 );
```

You can also match multiple ids in a single predicate:

Example 15.37: Matching all documents with an identifier among a given collection

```
List<Integer> ids = new ArrayList<>();
ids.add( 1 );
ids.add( 2 );
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.id().matchingAny( ids ) )
        .fetchHits( 20 );
```

Expected type of arguments

By default, the `id` predicate expects arguments to the `matching(…)`/`matchingAny(…)` methods to have the same type as the entity property corresponding to the document id.

For example, if the document identifier is generated from an entity identifier of type `Long`, the document identifier will still be of type `String`. `matching(…)`/`matchingAny(…)` will expect its argument to be of type `Long` regardless.

This should generally be what you want, but if you ever need to bypass conversion and pass an unconverted argument (of type `String`) to `matching(…)`/`matchingAny(…)`, see Type of arguments passed to the DSL.

Other options

- The score of an `id` predicate is constant and equal to 1 by default, but can be boosted with `.boost(…)`.

15.2.5. `match`: match a value

The `match` predicate matches documents for which a given field has a given value.

Example 15.38: Matching a value

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.match().field( "title" )
                .matching( "robot" ) )
        .fetchHits( 20 );
```

Expected type of arguments

By default, the `match` predicate expects arguments to the `matching(…)` method to have the same type as the entity property corresponding to the target field.

For example, if an entity property is of an enum type, the corresponding field may be of type `String`. `.matching(…)` will expect its argument to have the enum type regardless.

This should generally be what you want, but if you ever need to bypass conversion and pass an unconverted argument (of type `String` in the example above) to `.matching(…)`, see Type of arguments passed to the DSL.

Targeting multiple fields

Optionally, the predicate can target multiple fields. In that case, the predicate will match documents for which *any* of the given fields matches.

See Targeting multiple fields in one predicate.

Analysis

For most field types (number, date, …), the match is exact. However, for full-text fields or normalized keyword fields, the value passed to the `matching(…)` method is analyzed or normalized before being compared to the values in the index. This means the match is more subtle in two ways.

First, the predicate will not just match documents for which a given field has the exact same value: it will match all documents for which this field has a value whose normalized form is identical. See below for an example.

Example 15.39: Matching normalized terms

```
List<Author> hits = searchSession.search( Author.class )
        .where( f -> f.match().field( "lastName" )
                .matching( "ASIMOV" ) )①
        .fetchHits( 20 );

assertThat( hits ).extracting( Author::getLastName )
        .contains( "Asimov" );②
```

① For analyzed/normalized fields, the value passed to `matching` will be analyzed/normalized. In this case, the result of analysis is a lowercase string: `asimov`.

② All returned hits will have a value whose normalized form is identical. In this case, `Asimov` was normalized to `asimov` too, so it matched.

Second, for full-text fields, the value passed to the `matching(…)` method is tokenized. This means multiple terms may be extracted from the input value, and the predicate will match all documents for which the given field has a value that *contains any of those terms*, at any place and in any order. See below for an example.

Example 15.40: Matching multiple terms

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.match().field( "title" )
                .matching( "ROBOT Dawn" ) ) ①
        .fetchHits( 20 );

assertThat( hits ).extracting( Book::getTitle )
        .contains( "The Robots of Dawn", "I, Robot" ); ②
```

① For full-text fields, the value passed to `matching` will be tokenized and normalized. In this case, the result of analysis is the terms `robot` and `dawn` (notice they were lowercased).

② All returned hits will match **at least one** term of the given string. `The Robots of Dawn` contained both normalized terms `robot` and `dawn`, so it matched, but so did `I, Robot`, even though it didn't contain `dawn`: only one term is required.

Hits matching multiple terms, or matching more relevant terms, will have a higher score. Thus, if you sort by score, the most relevant hits will appear to the top of the result list. This usually makes up for the fact that the predicate does not require *all* terms to be present in matched documents.

If you need *all* terms to be present in matched documents, you should be able to do so by using the `simpleQueryString` predicate, in particular its ability to define a default operator. Just make sure to define which syntax features you want to expose to your users.

`fuzzy`: match a text value approximately

The `.fuzzy()` option allows for approximate matches, i.e. it allows matching documents for which a given field has a value that is not exactly the value passed to `matching(…)`, but a close value, for example with one letter that was switched for another.

This option is only available on text fields.

Example 15.41: Matching a text value approximately

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.match()
                .field( "title" )
                .matching( "robto" )
                .fuzzy() )
        .fetchHits( 20 );
```

Roughly speaking, the edit distance is the number of changes between two terms: switching characters, removing them, … It defaults to `2` when fuzzy matching is enabled, but can also be set to `0` (fuzzy matching disabled) or `1` (only one change allowed, so "less fuzzy"). Values higher than 2 are not allowed.

Example 15.42: Matching a text value approximately with explicit edit distance

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.match()
                .field( "title" )
                .matching( "robto" )
                .fuzzy( 1 ) )
        .fetchHits( 20 );
```

Optionally, you can force the match to be exact for the first `n` characters. `n` is called the "exact-prefix length". Setting this to a non-zero value is recommended for indexes containing a large amount of distinct terms, for performance reasons.

Example 15.43: Matching a text value approximately with exact prefix length

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.match()
                .field( "title" )
                .matching( "robto" )
                .fuzzy( 1, 3 ) )
        .fetchHits( 20 );
```

`minimumShouldMatch`: fine-tuning how many terms are required to match

It is possible to require that an arbitrary number of terms from the match string to be present in the document in order for the `match` predicate to match. This is the purpose of the `minimumShouldMatch*` methods, as demonstrated below.

Example 15.44: Fine-tuning matching requirements with `minimumShouldMatch`

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.match()
                .field( "title" )
                .matching( "investigation detective automatic" )
                .minimumShouldMatchNumber( 2 ) ) ①
        .fetchHits( 20 ); ②
```

① At least two terms must match for this predicate to match.

② All returned hits will match at least two of the terms: their titles will match either `investigation` and `detective`, `investigation` and `automatic`, `detective` and `automatic`, or all three of these terms.

Other options

- The score of a `match` predicate is variable for text fields by default, but can be made constant with `.constantScore()`.
- The score of a `match` predicate can be boosted, either on a per-field basis with a call to `.boost(…)` just after `.field(…)`/`.fields(…)` or for the whole predicate with a call to `.boost(…)` after `.matching(…)`.
- The `match` predicate uses the search analyzer of targeted fields to analyze searched text by default, but this can be overridden.

15.2.6. `range`: match a range of values

The `range` predicate matches documents for which a given field has a value within a given range.

Example 15.45: Matching a range of values

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.range().field( "pageCount" )
                .between( 210, 250 ) )
        .fetchHits( 20 );
```

The `between` method includes both bounds, i.e. documents whose value is exactly one of the bounds will match the `range` predicate.

At least one bound must be provided. If a bound is `null`, it will not constrain matches. For example `.between( 2, null )` will match all values higher than or equal to 2.

Different methods can be called instead of `between` in order to control the inclusion of the lower and upper bound:

`atLeast`

Example 206. Matching values equal to or greater than a given value

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.range().field( "pageCount" )
                .atLeast( 400 ) )
        .fetchHits( 20 );
```

`greaterThan`

Example 207. Matching values strictly greater than a given value

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.range().field( "pageCount" )
                .greaterThan( 400 ) )
        .fetchHits( 20 );
```

`atMost`

Example 208. Matching values equal to or less than a given value

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.range().field( "pageCount" )
                .atMost( 400 ) )
        .fetchHits( 20 );
```

`lessThan`

Example 209. Matching values strictly less than a given value

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.range().field( "pageCount" )
                .lessThan( 400 ) )
        .fetchHits( 20 );
```

Alternatively, you can specify whether bounds are included or excluded explicitly:

Example 15.46: Matching a range of values with explicit bound inclusion/exclusion

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.range().field( "pageCount" )
                .between(
                        200, RangeBoundInclusion.EXCLUDED,
                        250, RangeBoundInclusion.EXCLUDED
                ) )
        .fetchHits( 20 );
```

Sometimes it may be needed to match the value that is in one of the ranges. While it is possible to create an `or` predicate and add a `range` predicate for each of the ranges, there is a simpler way to do that:

Example 15.47: Matching values that are within any of the provided ranges

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.range().field( "pageCount" )
                .withinAny(
                        Range.between( 200, 250 ),
                        Range.between( 500, 800 )
                ) )
        .fetchHits( 20 );
```

Expected type of arguments

By default, the `range` predicate expects arguments to the `between(…)`/`atLeast(…)`/etc. method to have the same type as the entity property corresponding to the target field.

For example, if an entity property is of type `java.util.Date`, the corresponding field may be of type `java.time.Instant`; `between(…)`/`atLeast(…)`/etc. will expect its arguments to have type `java.util.Date` regardless. Similarly, `range(…)` will expect an argument of type `Range<java.util.Date>`.

This should generally be what you want, but if you ever need to bypass conversion and pass an unconverted argument (of type `java.time.Instant` in the example above) to `between(…)`/`atLeast(…)`/etc., see Type of arguments passed to the DSL.

Targeting multiple fields

Optionally, the predicate can target multiple fields. In that case, the predicate will match documents for which *any* of the given fields matches.

See Targeting multiple fields in one predicate.

Other options

- The score of a `range` predicate is constant and equal to 1 by default, but can be boosted, either on a per-field basis with a call to `.boost(…)` just after `.field(…)`/`.fields(…)` or for the whole predicate with a call to `.boost(…)` after `.between(…)`/`atLeast(…)`/etc.

15.2.7. `phrase`: match a sequence of words

The `phrase` predicate matches documents for which a given field contains a given sequence of words, in the given order.

This predicate is only available on full-text fields.

Example 15.48: Matching a sequence of words

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.phrase().field( "title" )
                .matching( "robots of dawn" ) )
        .fetchHits( 20 );
```

`slop`: match a sequence of words approximately

Specifying a *slop* allows for approximate matches, i.e. it allows matching documents for which a given field contains the given sequence of words, but in a slightly different order, or with extra words.

The slop represents the number of edit operations that can be applied to the sequence of words to match, where each edit operation moves one word by one position. So `quick fox` with a slop of `1` can become `quick <word> fox`, where `<word>` can be any word. `quick fox` with a slop of `2` can become `quick <word> fox`, or `quick <word1> <word2> fox` or even `fox quick` (two operations: moved `fox` to the left and `quick` to the right). And similarly for higher slops and for phrases with more words.

Example 15.49: Matching a sequence of words approximately with `slop(…)`

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.phrase().field( "title" )
                .matching( "dawn robot" )
                .slop( 3 ) )
        .fetchHits( 20 );
```

Targeting multiple fields

Optionally, the predicate can target multiple fields. In that case, the predicate will match documents for which *any* of the given fields matches.

See Targeting multiple fields in one predicate.

Other options

- The score of a `phrase` predicate is variable by default, but can be made constant with `.constantScore()`.
- The score of a `phrase` predicate can be boosted, either on a per-field basis with a call to `.boost(…)` just after `.field(…)`/`.fields(…)` or for the whole predicate with a call to `.boost(…)` after `.matching(…)`.
- The `phrase` predicate uses the search analyzer of targeted fields to analyze searched text by default, but this can be overridden.

15.2.8. `exists`: match fields with content

The `exists` predicate matches documents for which a given field has a non-null value.

Example 15.50: Matching fields with content

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.exists().field( "comment" ) )
        .fetchHits( 20 );
```

There isn't any built-in predicate to match documents for which a given field is null, but you can easily create one yourself by negating an `exists` predicate.

This can be achieved by passing in an `exists` predicate to a `not` predicate, or by using it in an `except` clause in a `matchAll` predicate.

Object fields

The `exists` predicate can also be applied to an object field. In that case, it will match all documents for which at least one inner field of the given object field has a non-null value.

Example 15.51: Matching object fields with content

```
List<Author> hits = searchSession.search( Author.class )
        .where( f -> f.exists().field( "placeOfBirth" ) )
        .fetchHits( 20 );
```

Object fields need to have at least one inner field with content in order to be considered as "existing".

Let's consider the example above, and let's assume the `placeOfBirth` object field only has one inner field: `placeOfBirth.country`:

- an author whose `placeOfBirth` is null will not match.
- an author whose `placeOfBirth` is not null and has the `country` filled in will match.
- an author whose `placeOfBirth` is not null but does not have the `country` filled in **will not match**.

Because of this, it is preferable to use the `exists` predicate on object fields that are known to have at least one inner field that is never null: an identifier, a name, …

Other options

- The score of an `exists` predicate is constant and equal to 1 by default, but can be boosted with a call to `.boost(…)`.

15.2.9. `wildcard`: match a simple pattern

The `wildcard` predicate matches documents for which a given field contains a word matching the given pattern.

Example 15.52: Matching a simple pattern

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.wildcard().field( "description" )
                .matching( "rob*t" ) )
        .fetchHits( 20 );
```

The pattern may include the following characters:

- `*` matches zero, one or multiple characters.
- `?` matches zero or one character.
- `\` escape the following character, e.g. `\?` is interpreted as a literal `?`, `\\` as a literal `\`, etc.
- any other character is interpreted as a literal.

If a normalizer has been defined on the field, the patterns used in wildcard predicates will be normalized.

If an analyzer has been defined on the field:

- when using the Elasticsearch backend, the patterns won't be analyzed nor normalized, and will be expected to match a **single** indexed token, not a sequence of tokens. This may behave differently on an older versions of the underlying search engine (for example with Elasticsearch 7.7-7.11 or OpenSearch prior to 2.5 the wildcard pattern will get normalized). Hence, please refer to the documentation of your particular version for the exact behaviour.
- when using the Lucene backend the patterns will be normalized, but not tokenized: the pattern will still be expected to match a **single** indexed token, not a sequence of tokens.

> For example, a pattern such as `Cat*` could match `cat` when targeting a field having a normalizer that applies a lowercase filter when indexing.
>
> A pattern such as `john gr*` will not match anything when targeting a field that tokenizes on spaces. `gr*` may match, since it doesn't include any space.
>
> When the goal is to match user-provided query strings, the simple query string predicate should be preferred.

Targeting multiple fields

Optionally, the predicate can target multiple fields. In that case, the predicate will match documents for which *any* of the given fields matches.

See Targeting multiple fields in one predicate.

Other options

- The score of a `wildcard` predicate is constant and equal to 1 by default, but can be boosted, either on a per-field basis with a call to `.boost(…)` just after `.field(…)`/`.fields(…)` or for the whole predicate with a call to `.boost(…)` after `.matching(…)`.

15.2.10. `regexp`: match a regular expression pattern

The `regexp` predicate matches documents for which a given field contains a word matching the given regular expression.

Example 15.53: Matching a regular expression pattern

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.regexp().field( "description" )
                .matching( "r.*t" ) )
        .fetchHits( 20 );
```

Regexp predicates and analysis

> For an introduction to analysis and how to configure it, refer to the Analysis section.

The behavior of regexp predicates on analyzed/normalized fields is a bit complex, so here is a summary of how it works.

Regexps must match the entirety of analyzed/normalized tokens

For a field that is lowercased and tokenized on spaces (using an analyzer), the regexp `robots?`:

- will match `Robot`: the indexed token `robot` matches.
- will match `I love robots`: the indexed token `robots` matches.
- will **not** match `Mr. Roboto`: the indexed token `roboto` does not match.

For a field that is lowercased but not tokenized (using a normalizer), the regexp `robots?`:

- will match `Robot`: the indexed token `robot` matches.
- will **not** match `I love robots`: the indexed token `i love robots` does not match.
- will **not** match `Mr. Roboto`: the indexed token `mr. roboto` does not match.

Regexps are never tokenized, even if fields are

Beware of spaces in regexps, in particular.

For a field that is tokenized on spaces (using an analyzer), the regexp `.*love .* robots?` will never match anything, because it requires a space inside the token and indexed tokens don't contain any (since tokenization happens on spaces).

For a field that is lowercased, but not tokenized (using a normalizer), the regexp `.*love .* robots?`:

- will match `I love robots`, which was indexed as `i love robots`.
- will match `I love my Robot`, which was indexed as `i love my robot`.
- will **not** match `I love Mr. Roboto`, which was indexed as `i love mr. roboto`: `roboto` doesn't match `robots?`.

With the Lucene backend, regexps are never analyzed nor normalized

For a field that is lowercased and tokenized on spaces:

- the regexp `Robots?` will **not** be normalized and will never match anything, because it requires an uppercase letter and indexed tokens don't contain any (since they are lowercased).
- the regexp `[Rr]obots?` will **not** be normalized but will match `I love Robots`: the indexed token `robots` matches.
- the regexp `love .* robots?` will **not** be normalized and will match `I love my Robot` as well as `I love robots`, but not `Robots love me`.

With the Elasticsearch backend, regexps are not analyzed nor normalized on text (tokenized) fields, but are normalized on keyword (non-tokenized) fields

For a field that is lowercased and tokenized on spaces (using an analyzer):

- the regexp `Robots?` will **not** be normalized and will never match anything, because it requires an uppercase letter and indexed tokens don't contain any (since they are lowercased).
- the regexp `[Rr]obots?` will **not** be normalized but will match `I love Robots`: the indexed token `robots` matches.
- the regexp `love .* robots?` will **not** be normalized and will match `I love my Robot` as well as `I love robots`, but not `Robots love me`.

However, **behavior differs from Lucene for normalized fields**! For a field that is lowercased, but not tokenized (using a normalizer):

- the regexp `Robots?+` will be normalized to `robots?` and will match `I love robots`: the indexed token `robots` matches.
- the regexp `[Rr]obots?+` will be normalized to `[rr]obots?` and will match `I love Robots`: the indexed token `robots` matches.
- the regexp `love .* robots?` will match `I love my Robot` as well as `I love robots`, but not `Robots love me`.

As a result of Elasticsearch normalizing regular expressions, normalizers can interfere with regexp meta-characters and completely change the meaning of a regexp.

For example, for a field whose normalizer replaces the characters `*` and `?` with `_`, the regexp `Robots?` will be normalized to `Robots_` and will probably never match anything.

This behavior is considered a bug and was reported to the Elasticsearch project.

`flags`: enabling only specific syntax constructs

By default, Hibernate Search does not enable any optional operators. To enable some of them, it is possible to specify the `flags` attribute.

Example 15.54: Matching a regular expression pattern with flags

```
hits = searchSession.search( Book.class )
        .where( f -> f.regexp().field( "description" )
                .matching( "r@t" )
                .flags( RegexpQueryFlag.ANY_STRING )
        )
        .fetchHits( 20 );
```

The following flags/operators are available:

- `INTERVAL`: the `<>` operator matches a non-negative integer range, both ends included.

  For example `a<1-10>` matches `a1`, `a2`, … `a9`, `a10`, but not `a11`.

  Leading zeroes are meaningful, e.g. `a<01-10>` matches `a01` and `a02` but not `a1` nor `a2`.

- `INTERSECTION`: the `&` operator combines two regexps with an AND operator.

  For example `.*a.*&.*z.*` matches `az`, `za`, `babzb`, `bzbab`, but not `a` nor `z`.

- `ANYSTRING`: the `@` operator matches any string; equivalent to `.*`.

  This operator is mostly useful to negate a pattern, e.g. `@&~(ab)` matches anything except the string `ab`.

Targeting multiple fields

Optionally, the predicate can target multiple fields. In that case, the predicate will match documents for which *any* of the given fields matches.

See Targeting multiple fields in one predicate.

Other options

- The score of a `regexp` predicate is constant and equal to 1 by default, but can be boosted, either on a per-field basis with a call to `.boost(…)` just after `.field(…)`/`.fields(…)` or for the whole predicate with a call to `.boost(…)` after `.matching(…)`.

15.2.11. `terms`: match a set of terms

The `terms` predicate matches documents for which a given field contains some terms, any or all of them.

With `matchingAny` we require that at least one of the provided terms matches. Functionally, this is somewhat similar to a boolean `OR` with one `match` predicate per term, but the syntax for a single `terms` predicate is more concise.

`matchingAny` expects to be passed **terms**, not just any string. The given terms will not be analyzed. See `terms` predicates and analysis.

Example 15.55: Matching any of the provided terms

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.terms().field( "genre" )
                .matchingAny( Genre.CRIME_FICTION, Genre.SCIENCE_FICTION ) )
        .fetchHits( 20 );
```

With `matchingAll` we require that all the provided terms match. Functionally, this is somewhat similar to a boolean `AND` with one `match` predicate per term, but the syntax for a single `terms` predicate is more concise.

`matchingAll` expects to be passed **terms**, not just any string. The given terms will not be analyzed. See `terms` predicates and analysis.

By default, `matchingAll` will not accept more than 1024 terms.

It is possible to raise this limit through backend-specific configuration:

- For the Lucene backend, run this code when starting up your application:
  `org.apache.lucene.search.BooleanQuery.maxClauseCount = <your limit>;`
- For the Elasticsearch backend, see the global setting `indices.query.bool.max_clause_count` in the Elasticsearch documentation.

However, keep in mind the limit is there for a reason: attempts to match very large numbers of terms will perform poorly and could lead to crashes.

Example 15.56: Matching all the provided terms

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.terms().field( "genre" )
                .matchingAny( Genre.CRIME_FICTION, Genre.SCIENCE_FICTION ) )
        .fetchHits( 20 );
```

`terms` predicates and analysis

For an introduction to analysis and how to configure it, refer to the Analysis section.

Differently from other predicates, the terms passed to `matchingAny()` or `matchingAll()` are never analyzed nor **usually** normalized.

If an analyzer has been defined on the field, the terms will not be analyzed nor normalized.

If a normalizer has been defined on the field:

- when using the Elasticsearch backend, the terms **will** be normalized.
- when using the Lucene backend, the terms **will not** be normalized.

For example, the term `Cat` could match `cat` when targeting a field having a normalizer that applies a lowercase filter when

> indexing, but only when using the Elasticsearch backend. When using the Lucene backend, only the term `cat` could match `cat`.

Expected type of arguments

By default, the `terms` predicate expects arguments to the `matchingAny(…)` or `matchingAll(…)` methods to have the same type as the entity property corresponding to the target field.

For example, if an entity property is of an enum type, the corresponding field may be of type `String`. `.matchingAny(…)` will expect its argument to have the enum type regardless.

This should generally be what you want, but if you ever need to bypass conversion and pass an unconverted argument (of type `String` in the example above) to `.matchingAny(…)` or `.matchingAll(…)`, see Type of arguments passed to the DSL.

Targeting multiple fields

Optionally, the predicate can target multiple fields. In that case, the predicate will match documents for which *any* of the given fields matches.

See Targeting multiple fields in one predicate.

Other options

- The score of a `terms` predicate is constant and equal to 1 by default, but can be boosted, either on a per-field basis with a call to `.boost(…)` just after `.field(…)`/`.fields(…)` or for the whole predicate with a call to `.boost(…)` after `.matchingAny(…)` or `.matchingAll(…)`.

15.2.12. `and`: match all clauses

The `and` predicate matches documents that match **all** of its inner predicates, called "clauses".

Matching "and" clauses are taken into account during score computation.

Example 15.57: Matching a document that matches all the multiple given predicates (`~AND` operator)

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.and(
                f.match().field( "title" )
                        .matching( "robot" ), ①
                f.match().field( "description" )
                        .matching( "crime" ) ②
        ) )
        .fetchHits( 20 ); ③
```

① The hits **must** have a `title` field matching the text `robot`, independently of other clauses in the same boolean predicate.
② The hits **must** have a `description` field matching the text `crime`, independently of other clauses in the same boolean predicate.
③ All returned hits will match **all** the clauses above: they will have a `title` field matching the text `robot` **and** they will have a `description` field matching the text `crime`.

Adding clauses dynamically with the lambda syntax

It is possible to define the `and` predicate inside a lambda expression. This is especially useful when clauses need to be added dynamically to the `and` predicate, for example based on user input.

Example 15.58: Easily adding clauses dynamically using `.where(…)` and the lambda syntax

```
MySearchParameters searchParameters = getSearchParameters(); ①
List<Book> hits = searchSession.search( Book.class )
        .where( (f, root) -> { ②
            root.add( f.matchAll() ); ③
            if ( searchParameters.getGenreFilter() != null ) { ④
                root.add( f.match().field( "genre" )
                        .matching( searchParameters.getGenreFilter() ) );
            }
            if ( searchParameters.getFullTextFilter() != null ) {
```

```
                root.add( f.match().fields( "title", "description" )
                        .matching( searchParameters.getFullTextFilter() ) );
            }
            if ( searchParameters.getPageCountMaxFilter() != null ) {
                root.add( f.range().field( "pageCount" )
                        .atMost( searchParameters.getPageCountMaxFilter() ) );
            }
        } )
        .fetchHits( 20 );
```

① Get a custom object holding the search parameters provided by the user through a web form, for example.

② Call `.where(BiConsumer)`. The consumer, implemented by a lambda expression, will receive a predicate factory as well as clause collector as an argument, and will add clauses to that collector as necessary.

③ By default, a boolean predicate will match nothing if there is no clause. To match every document when there is no clause, add a `and` clause that matches everything.

④ Inside the lambda, the code is free to use any Java language constructs, such as `if` or `for`, to control the addition of clauses. In this case, we only add clauses if the relevant parameter was filled in by the user.

Another syntax relying on the method `with(…)` can be useful when the `and` predicate is not the root predicate:

Example 15.59: Easily adding clauses dynamically using `with(…)` and the lambda syntax

```
MySearchParameters searchParameters = getSearchParameters(); ①
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.and().with( and -> { ②
            and.add( f.matchAll() ); ③
            if ( searchParameters.getGenreFilter() != null ) { ④
                and.add( f.match().field( "genre" )
                        .matching( searchParameters.getGenreFilter() ) );
            }
            if ( searchParameters.getFullTextFilter() != null ) {
                and.add( f.match().fields( "title", "description" )
                        .matching( searchParameters.getFullTextFilter() ) );
            }
            if ( searchParameters.getPageCountMaxFilter() != null ) {
                and.add( f.range().field( "pageCount" )
                        .atMost( searchParameters.getPageCountMaxFilter() ) );
            }
        } ) )
        .fetchHits( 20 );
```

① Get a custom object holding the search parameters provided by the user through a web form, for example.

② Call `.where(Function)`. The function, implemented by a lambda expression, will receive a predicate factory, use it to build an `and` predicate, invoke the `with(Consumer)` method and return this predicate. The consumer, implemented by a lambda expression, will receive a clause collector for the `and` predicate as an argument, and will add clauses to that collector as necessary.

③ By default, a boolean predicate will match nothing if there is no clause. To match every document when there is no clause, add an `and` clause that matches everything.

④ Inside the lambda, the code is free to use any Java language constructs, such as `if` or `for`, to control the addition of clauses. In this case, we only add clauses if the relevant parameter was filled in by the user.

Options

- The score of an `and` predicate is variable by default, but can be made constant with `.constantScore()`.
- The score of an `and` predicate can be boosted with a call to `.boost(…)`.

15.2.13. `or`: match any clause

The `or` predicate matches documents that match **any** of its inner predicates, called "clauses".

Matching `or` clauses are taken into account during score computation.

Example 15.60: Matching a document that matches any of multiple given predicates (`~OR` operator)

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.or(
                f.match().field( "title" )
                        .matching( "robot" ), ①
                f.match().field( "description" )
                        .matching( "investigation" ) ②
        ) )
        .fetchHits( 20 ); ③
```

① The hits **should** have a `title` field matching the text `robot`, **or** they should match any other clause in the same boolean predicate.

② The hits **should** have a `description` field matching the text `investigation`, **or** they should match any other clause in the same boolean predicate.

③ All returned hits will match **at least one** of the clauses above: they will have a `title` field matching the text `robot` **or** they will have a `description` field matching the text `investigation`.

Adding clauses dynamically with the lambda syntax

It is possible to define the `or` predicate inside a lambda expression. This is especially useful when clauses need to be added dynamically to the `or` predicate, for example based on user input.

Example 15.61: Easily adding clauses dynamically using `with(…)` and the lambda syntax

```
MySearchParameters searchParameters = getSearchParameters(); ①
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.or().with( or -> { ②
            if ( !searchParameters.getAuthorFilters().isEmpty() ) {
                for ( String authorFilter : searchParameters.getAuthorFilters() ) { ③
                    or.add( f.match().fields( "authors.firstName", "authors.lastName" )
                            .matching( authorFilter ) );
                }
            }
        } ) )
        .fetchHits( 20 );
```

① Get a custom object holding the search parameters provided by the user through a web form, for example.0

② Call `.where(Function)`. The function, implemented by a lambda expression, will receive a predicate factory, use it to build an `or` predicate, invoke the `with(Consumer)` method and return this predicate. The consumer, implemented by a lambda expression, will receive a clause collector for the `or` predicate as an argument, and will add clauses to that collector as necessary.

③ Inside the lambda, the code is free to use any Java language constructs, such as `if` or `for`, to control the addition of clauses. In this case, we only add clauses if the relevant parameter was filled in by the user.

Options

- The score of an `or` predicate is variable by default, but can be made constant with `.constantScore()`.
- The score of an `or` predicate can be boosted with a call to `.boost(…)`.

15.2.14. `not`: negating another predicate

The `not` predicate matches documents that are not matched by a given predicate.

Example 15.62: Negating a `match` predicate

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.not(
                f.match()
                        .field( "genre" )
                        .matching( Genre.SCIENCE_FICTION )
        ) )
        .fetchHits( 20 );
```

Other options

- The score of a `not` predicate is constant and equal to 0 by default, but if boosted with `.boost(…)` the default would be changed to 1 and corresponding boost will be applied.

15.2.15. `bool`: advanced combinations of predicates (or/and/…)

The `bool` predicate allows combining inner predicates in a more complex fashion than with simpler `and`/`or` predicates.

The `bool` predicate matches documents that match one or more inner predicates, called "clauses". It can be used in particular to build `AND`/`OR` operators with additional settings.

Inner predicates are added as clauses of one of the following types:

`must`

: `must` clauses are required to match: if they don't match, then the `bool` predicate will not match.

  Matching "must" clauses are taken into account during score computation.

`mustNot`

: `mustNot` clauses are required to not match: if they match, then the `bool` predicate will not match.

  "must not" clauses are ignored during score computation.

`filter`

: `filter` clauses are required to match: if they don't match, then the boolean predicate will not match.

  `filter` clauses are ignored during score computation, and so are any clauses of boolean predicates contained in the filter clause (even `must` or `should` clauses).

`should`

: `should` clauses may optionally match, and are required to match depending on the context.

  Matching `should` clauses are taken into account during score computation.

  The exact behavior of `should` clauses is as follows:

  - When there isn't any `must` clause nor any `filter` clause in the `bool` predicate then at least one "should" clause is required to match. Simply put, in this case, the "should" clauses behave as if there was an `OR` operator between each of them.
  - When there is at least one `must` clause or one `filter` clause in the `bool` predicate, then the "should" clauses are not required to match, and are simply used for scoring.
  - This behavior can be changed by specifying `minimumShouldMatch` constraints.

Emulating an `OR` operator

A `bool` predicate with only `should` clauses and no `minimumShouldMatch` specification will behave as an `OR` operator. In such case, using the simpler `or` syntax is recommended.

Emulating an `AND` operator

A `bool` predicate with only `must` clauses will behave as an `AND` operator. In such case, using the simpler `and` syntax is recommended.

`mustNot`: excluding documents that match a given predicate

Example 15.63: Matching a document that does ***not*** *match a given predicate*

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.bool()
                .must( f.match().field( "title" )
                        .matching( "robot" ) ) ①
                .mustNot( f.match().field( "description" )
                        .matching( "investigation" ) ) ②
        )
        .fetchHits( 20 ); ③
```

① The hits **must** have a `title` field matching the text `robot`, independently of other clauses in the same boolean predicate.

② The hits **must not** have a `description` field matching the text `investigation`, independently of other clauses in the same boolean predicate.
③ All returned hits will match **all** the clauses above: they will have a `title` field matching the text `robot` **and** they will not have a `description` field matching the text `investigation`.

While it is possible to execute a boolean predicate with only "negative" clauses (`mustNot`), performance may be disappointing because the full power of indexes cannot be leveraged in that case.

`filter`: matching documents that match a given predicate without affecting the score

`filter` clauses are essentially `must` clauses with only one difference: they are ignored when computing the total score of a document.

Example 15.64: Matching a document that matches a given predicate without affecting the score

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.bool() ①
                .should( f.bool() ②
                        .filter( f.match().field( "genre" )
                                .matching( Genre.SCIENCE_FICTION ) ) ③
                        .must( f.match().fields( "description" )
                                .matching( "crime" ) ) ④
                )
                .should( f.bool() ⑤
                        .filter( f.match().field( "genre" )
                                .matching( Genre.CRIME_FICTION ) ) ⑥
                        .must( f.match().fields( "description" )
                                .matching( "robot" ) ) ⑦
                )
        )
        .fetchHits( 20 ); ⑧
```

① Create a top-level boolean predicate, with two `should` clauses.
② In the first `should` clause, create a nested boolean predicate.
③ Use a `filter` clause to require documents to have the `science-fiction` genre, without taking this predicate into account when scoring.
④ Use a `must` clause to require documents with the `science-fiction` genre to have a `title` field matching `crime`, and take this predicate into account when scoring.
⑤ In the second `should` clause, create a nested boolean predicate.
⑥ Use a `filter` clause to require documents to have the `crime fiction` genre, without taking this predicate into account when scoring.
⑦ Use a `must` clause to require documents with the `crime fiction` genre to have a `description` field matching `robot`, and take this predicate into account when scoring.
⑧ The score of hits will ignore the `filter` clauses, leading to fairer sorts if there are much more "crime fiction" documents than "science-fiction" documents.

`should` as a way to tune scoring

Apart from being used alone to emulate an `OR` operator, `should` clauses can also be used in conjunction with `must` clauses. When doing so, the `should` clauses become completely optional, and their only purpose is to increase the score of documents that match these clauses.

Example 15.65: Using optional `should` clauses to boost the score of some documents

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.bool()
                .must( f.match().field( "title" )
                        .matching( "robot" ) ) ①
                .should( f.match().field( "description" )
                        .matching( "crime" ) ) ②
                .should( f.match().field( "description" )
                        .matching( "investigation" ) ) ③
        )
        .fetchHits( 20 ); ④
```

① The hits **must** have a `title` field matching the text `robot`, independently of other clauses in the same boolean predicate.
② The hits **should** have a `description` field matching the text `crime`, but they might not, because matching the `must` clause above is enough. However, matching this `should` clause will improve the score of the document.
③ The hits **should** have a `description` field matching the text `investigation`, but they might not, because matching the `must` clause above is enough. However, matching this `should` clause will improve the score of the document.
④ All returned hits will match the `must` clause, and optionally the `should` clauses: they will have a `title` field matching the text `robot`, and the ones whose description matches either `crime` or `investigation` will have a better score.

`minimumShouldMatch`: fine-tuning how many `should` clauses are required to match

It is possible to require that an arbitrary number of `should` clauses match in order for the `bool` predicate to match. This is the purpose of the `minimumShouldMatch*` methods, as demonstrated below.

Example 15.66: Fine-tuning `should` clauses matching requirements with `minimumShouldMatch`

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.bool()
                .minimumShouldMatchNumber( 2 ) ①
                .should( f.match().field( "description" )
                        .matching( "robot" ) ) ②
                .should( f.match().field( "description" )
                        .matching( "investigation" ) ) ③
                .should( f.match().field( "description" )
                        .matching( "disappearance" ) ) ④
        )
        .fetchHits( 20 ); ⑤
```

① At least two "should" clauses must match for this boolean predicate to match.
② The hits **should** have a `description` field matching the text `robot`.
③ The hits **should** have a `description` field matching the text `investigate`.
④ The hits **should** have a `description` field matching the text `crime`.
⑤ All returned hits will match at least two of the `should` clauses: their description will match either `robot` and `investigate`, `robot` and `crime`, `investigate` and `crime`, or all three of these terms.

Adding clauses dynamically with the lambda syntax

It is possible to define the `bool` predicate inside a lambda expression. This is especially useful when clauses need to be added dynamically to the `bool` predicate, for example based on user input.

If you simply want to build a root predicate matching multiple, dynamically generated clauses, consider using the `.where( (f, root) → … )` syntax instead.

Example 15.67: Easily adding clauses dynamically using `with(…)` and the lambda syntax

```
MySearchParameters searchParameters = getSearchParameters(); ①
List<Book> hits = searchSession.search( Book.class )
        .where( (f, root) -> { ②
            root.add( f.matchAll() );
            if ( searchParameters.getGenreFilter() != null ) {
                root.add( f.match().field( "genre" )
                        .matching( searchParameters.getGenreFilter() ) );
            }
            if ( !searchParameters.getAuthorFilters().isEmpty() ) {
                root.add( f.bool().with( b -> { ③
                    for ( String authorFilter : searchParameters.getAuthorFilters() ) { ④
                        b.should( f.match().fields( "authors.firstName", "authors.lastName" )
                                .matching( authorFilter ) );
                    }
                } ) );
            }
        } )
        .fetchHits( 20 );
```

① Get a custom object holding the search parameters provided by the user through a web form, for example.
② Call `.where(BiConsumer)` to create the root predicate, which is not the one we're interested in here.
③ Call `.bool().with(Consumer)` to create an inner, non-root predicate. The consumer, implemented by a lambda expression, will receive a collector as an argument and will add clauses to that collector as necessary.
④ Inside the lambda, the code is free to use any Java language constructs, such as `if` or `for`, to control the addition of clauses. In this case, we add one clause per author filter.

Deprecated variants

Features detailed in this section are *deprecated*: they should be avoided in favor of non-deprecated alternatives.

The usual compatibility policy applies, meaning the features are expected to remain available at least until the next major version of Hibernate Search. Beyond that, they may be altered in a backward-incompatible way—or even removed.

Usage of deprecated features is not recommended.

Another syntax can be used to create a boolean predicate from a lambda expression, but it is deprecated.

Example 15.68: Deprecated variant of `.bool`

```
MySearchParameters searchParameters = getSearchParameters(); ①
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.bool( b -> { ②
            b.must( f.matchAll() ); ③
            if ( searchParameters.getGenreFilter() != null ) { ④
                b.must( f.match().field( "genre" )
                        .matching( searchParameters.getGenreFilter() ) );
            }
            if ( searchParameters.getFullTextFilter() != null ) {
                b.must( f.match().fields( "title", "description" )
                        .matching( searchParameters.getFullTextFilter() ) );
            }
            if ( searchParameters.getPageCountMaxFilter() != null ) {
                b.must( f.range().field( "pageCount" )
                        .atMost( searchParameters.getPageCountMaxFilter() ) );
            }
        } ) )
        .fetchHits( 20 );
```

① Get a custom object holding the search parameters provided by the user through a web form, for example.
② Call `.bool(Consumer)`. The consumer, implemented by a lambda expression, will receive a collector as an argument and will add clauses to that collector as necessary.
③ By default, a boolean predicate will match nothing if there is no clause. To match every document when there is no clause, add a `must` clause that matches everything.
④ Inside the lambda, the code is free to use any Java language constructs, such as `if` or `for`, to control the addition of clauses. In this case, we only add clauses if the relevant parameter was filled in by the user.

Other options

- The score of a `bool` predicate is variable by default, but can be made constant with `.constantScore()`.
- The score of a `bool` predicate can be boosted with a call to `.boost(…)`.

15.2.16. `simpleQueryString`: match a user-provided query string

The `simpleQueryString` predicate matches documents according to a structured query given as a string.

Its syntax is quite simple, so it's especially helpful when end user expect to be able to submit text queries with a few syntax elements such as boolean operators, quotes, etc.

Boolean operators

The syntax includes three boolean operators:

- AND using `+`
- OR using `|`
- NOT using `-`

Example 15.69: Matching a simple query string: AND/OR operators

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.simpleQueryString().field( "description" )
                .matching( "robots + (crime | investigation | disappearance)" ) )
        .fetchHits( 20 );
```

Example 15.70: Matching a simple query string: NOT operator

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.simpleQueryString().field( "description" )
                .matching( "robots + -investigation" ) )
        .fetchHits( 20 );
```

Default boolean operator

By default, the query uses the OR operator if the operator is not explicitly defined. If you prefer using the AND operator as default, you can call `.defaultOperator(…)`.

Example 15.71: Matching a simple query string: AND as default operator

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.simpleQueryString().field( "description" )
                .matching( "robots investigation" )
                .defaultOperator( BooleanOperator.AND ) )
        .fetchHits( 20 );
```

Prefix

The syntax includes support for prefix predicates through the `*` wildcard.

Example 15.72: Matching a simple query string: prefix

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.simpleQueryString().field( "description" )
                .matching( "rob*" ) )
        .fetchHits( 20 );
```

The `*` wildcard will only be understood at the end of a word. `rob*t` will be interpreted as a literal. This really is a *prefix* predicate, not a *wildcard* predicate.

Fuzzy

The syntax includes support for the fuzzy operator `~`. Its behavior is similar to that of fuzzy matching in the `match` predicate.

Example 15.73: Matching a simple query string: fuzzy

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.simpleQueryString().field( "description" )
                .matching( "robto~2" ) )
```

```
        .fetchHits( 20 );
```

Phrase

The syntax includes support for `phrase` predicates using quotes around the sequence of terms to match.

Example 15.74: Matching a simple query string: phrase

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.simpleQueryString().field( "title" )
                .matching( "\"robots of dawn\"" ) )
        .fetchHits( 20 );
```

A slop can be assigned to a phrase predicate using the NEAR operator `~`.

Example 15.75: Matching a simple query string: phrase with slop

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.simpleQueryString().field( "title" )
                .matching( "\"dawn robot\"~3" ) )
        .fetchHits( 20 );
```

`flags`: enabling only specific syntax constructs

By default, all syntax features are enabled. You can pick the operators to enable explicitly through the `.flags(…)` method.

Example 15.76: Matching a simple query string: enabling only specific syntax constructs

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.simpleQueryString().field( "title" )
                .matching( "I want a **robot**" )
                .flags( SimpleQueryFlag.AND, SimpleQueryFlag.OR, SimpleQueryFlag.NOT ) )
        .fetchHits( 20 );
```

If you wish, you can disable all syntax constructs:

Example 15.77: Matching a simple query string: disabling all syntax constructs

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.simpleQueryString().field( "title" )
                .matching( "**robot**" )
                .flags( Collections.emptySet() ) )
        .fetchHits( 20 );
```

`minimumShouldMatch`: fine-tuning how many `should` clauses are required to match

The resulting query parsed from the query string may result in a boolean query with should clauses. It may be helpful to control how many `should` clauses must match to consider a document as a match.

Example 15.78: Fine-tuning `should` clauses matching requirements with `minimumShouldMatch`

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.simpleQueryString().field( "title" )
                .matching( "crime robot investigate automatic detective" )
                .minimumShouldMatchNumber( 2 ) )
        .fetchHits( 20 );
```

This is similar to boolean or query string predicates `minimumShouldMatch` options.

Targeting multiple fields

Optionally, the predicate can target multiple fields. In that case, the predicate will match documents for which *any* of the given fields matches.

See Targeting multiple fields in one predicate.

If targeted fields have different analyzers, an exception will be thrown. You can avoid this by picking an analyzer explicitly, but make sure you know what you're doing.

Field types and expected format of field values

This predicate is applicable to most of the supported field types except `GeoPoint` and vector field types.

The format of string literals used in the query string is backend-specific. With the Lucene backend, the format of these literals should be compatible with the parsing logic defined in Property types with built-in value bridges, and for fields with custom bridges it must be defined . As for the Elasticsearch backend, see the Field types supported by the Elasticsearch backend.

Keep in mind that **not** all query constructs can be applied to non-string fields, e.g. adding fuzziness, slop or wildcards will not work.

Other options

- The score of a `simpleQueryString` predicate is variable by default, but can be made constant with `.constantScore()`.
- The score of a `simpleQueryString` predicate can be boosted, either on a per-field basis with a call to `.boost(…)` just after `.field(…)` / `.fields(…)` or for the whole predicate with a call to `.boost(…)` after `.matching(…)`.
- The `simpleQueryString` predicate uses the search analyzer of targeted fields to analyze searched text by default, but this can be overridden.

15.2.17. `nested`: match nested documents

The `nested` predicate can be used on object fields indexed as nested documents to require two or more inner predicates to match *the same object*. This is how you ensure that `authors.firstname:isaac AND authors.lastname:asimov` will not match a book whose authors are "Jane Asimov" and "Isaac Deutscher".

Example 15.79: Matching multiple predicates against a single nested object

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.nested( "authors" ) ①
                .add( f.match().field( "authors.firstName" )
                        .matching( "isaac" ) ) ②
                .add( f.match().field( "authors.lastName" )
                        .matching( "asimov" ) ) ) ③
        .fetchHits( 20 ); ④
```

① Create a nested predicate on the `authors` object field.

② The author must have a first name matching `isaac`.

③ The author must have a last name matching `asimov`.

④ All returned hits will be books for which at least one author has a first name matching `isaac` and a last name matching `asimov`. Books that happen to have multiple authors, one of which has a first name matching `isaac` and **another** of which has a last name matching `asimov`, will **not** match.

Implicit nesting

Hibernate Search automatically wraps a nested predicate around other predicates when necessary. However, this is done for each single predicate, so implicit nesting will not give the same behavior as explicit nesting grouping multiple inner predicates. See below for an example.

Example 15.80: Using implicit nesting

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.and()
```

```
                .add( f.match().field( "authors.firstName" ) ①
                        .matching( "isaac" ) ) ②
                .add( f.match().field( "authors.lastName" )
                        .matching( "asimov" ) ) ) ③
        .fetchHits( 20 ); ④
```

① The nested predicate is created implicitly, since target fields here belong to a nested object.
② The author must have a first name matching `isaac`.
③ The author must have a last name matching `asimov`.
④ All returned hits will be books for which at least one author has a first name matching `isaac` and a last name matching `asimov`. Books that happen to have multiple authors, one of which has a first name matching `isaac` and **another** of which has a last name matching `asimov`, will match, because we apply the nested predicate **separately** to **each match** predicate.

Deprecated variants

Features detailed in this section are *deprecated*: they should be avoided in favor of non-deprecated alternatives.

The usual compatibility policy applies, meaning the features are expected to remain available at least until the next major version of Hibernate Search. Beyond that, they may be altered in a backward-incompatible way—or even removed.

Usage of deprecated features is not recommended.

Another syntax can be used to create a nested predicate, but it is more verbose and deprecated.

Example 15.81: Deprecated variant of `.nested`

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.nested().objectField( "authors" ) ①
                .nest( f.and()
                        .add( f.match().field( "authors.firstName" )
                                .matching( "isaac" ) ) ②
                        .add( f.match().field( "authors.lastName" )
                                .matching( "asimov" ) ) ) ) ③
        .fetchHits( 20 ); ④
```

① Create a nested predicate on the `authors` object field.
② The author must have a first name matching `isaac`.
③ The author must have a last name matching `asimov`.
④ All returned hits will be books for which at least one author has a first name matching `isaac` and a last name matching `asimov`. Books that happen to have multiple authors, one of which has a first name matching `isaac` and **another** of which has a last name matching `asimov`, will **not** match.

15.2.18. `within`: match points within a circle, box, polygon

The `within` predicate matches documents for which a given field is a geo-point contained within a given circle, bounding-box or polygon.

This predicate is only available on geo-point fields.

Matching points within a circle (within a distance to a point)

With `.circle(…)`, the matched points must be within a given distance from a given point (center).

Example 15.82: Matching points within a circle

```
GeoPoint center = GeoPoint.of( 53.970000, 32.150000 );
List<Author> hits = searchSession.search( Author.class )
        .where( f -> f.spatial().within().field( "placeOfBirth.coordinates" )
                .circle( center, 50, DistanceUnit.KILOMETERS ) )
        .fetchHits( 20 );
```

Other distance units are available, in particular `METERS`, `YARDS` and `MILES`. When the distance unit is omitted, it defaults to `METERS`.

You can also pass the coordinates of the center as two doubles (latitude, then longitude).

Example 15.83: Matching points within a circle: passing center coordinates as doubles

```
List<Author> hits = searchSession.search( Author.class )
        .where( f -> f.spatial().within().field( "placeOfBirth.coordinates" )
                .circle( 53.970000, 32.150000, 50, DistanceUnit.KILOMETERS ) )
        .fetchHits( 20 );
```

Matching points within a bounding box

With `.boundingBox(…)`, the matched points must be within a given bounding box defined by its top-left and bottom-right corners.

Example 15.84: Matching points within a box

```
GeoBoundingBox box = GeoBoundingBox.of(
        53.99, 32.13,
        53.95, 32.17
);
List<Author> hits = searchSession.search( Author.class )
        .where( f -> f.spatial().within().field( "placeOfBirth.coordinates" )
                .boundingBox( box ) )
        .fetchHits( 20 );
```

You can also pass the coordinates of the top-left and bottom-right corners as four doubles: top-left latitude, top-left longitude, bottom-right latitude, bottom-right longitude.

Example 15.85: Matching points within a box: passing corner coordinates as doubles

```
List<Author> hits = searchSession.search( Author.class )
        .where( f -> f.spatial().within().field( "placeOfBirth.coordinates" )
                .boundingBox( 53.99, 32.13,
                        53.95, 32.17 ) )
        .fetchHits( 20 );
```

Matching points within a polygon

With `.polygon(…)`, the matched points must be within a given polygon.

Example 15.86: Matching points within a polygon

```
GeoPolygon polygon = GeoPolygon.of(
        GeoPoint.of( 53.976177, 32.138627 ),
        GeoPoint.of( 53.986177, 32.148627 ),
        GeoPoint.of( 53.979177, 32.168627 ),
        GeoPoint.of( 53.876177, 32.159627 ),
        GeoPoint.of( 53.956177, 32.155627 ),
        GeoPoint.of( 53.976177, 32.138627 )
);
List<Author> hits = searchSession.search( Author.class )
        .where( f -> f.spatial().within().field( "placeOfBirth.coordinates" )
                .polygon( polygon ) )
        .fetchHits( 20 );
```

Targeting multiple fields

Optionally, the predicate can target multiple fields. In that case, the predicate will match documents for which *any* of the given fields matches.

See Targeting multiple fields in one predicate.

Other options

- The score of a `within` predicate is constant and equal to 1 by default, but can be boosted, either on a per-field basis with a call to `.boost(…)` just after `.field(…)`/`.fields(…)` or for the whole predicate with a call to `.boost(…)` after `.circle(…)`/`.boundingBox(…)`/`.polygon(…)`.

15.2.19. `knn`: K-Nearest Neighbors a.k.a. vector search

> Features detailed below are *incubating*: they are still under active development.
>
> The usual compatibility policy does not apply: the contract of incubating elements (e.g. types, methods, configuration properties, etc.) may be altered in a backward-incompatible way — or even removed — in subsequent releases.
>
> You are encouraged to use incubating features so the development team can get feedback and improve them, but you should be prepared to update code which relies on them as needed.

The `knn` predicate, with `k` being a positive integer, matches the `k` documents for which a given vector field's value is "nearest" to a given vector.

Distance is measured based on the vector similarity configured for the given vector field.

Example 15.87: Simple K-Nearest Neighbors search

```
float[] coverImageEmbeddingsVector = /*...*/
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.knn( 5 ).field( "coverImageEmbeddings" ).matching( coverImageEmbeddingsVector ) )
        .fetchHits( 20 );
```

Expected type of arguments

The `knn` predicate expects arguments to the `matching(…)` method to have the same type as the index type of a target field.

For example, if an entity property is mapped in the index to a byte array type (`byte[]`) , `.matching(…)` will expect its argument to be a byte array (`byte[]`) only.

Filtering the neighbors

Optionally, the predicate can filter out some of the neighbors using the `.filter(..)` clause of the predicate. `.filter(…)` expects a predicate to be passed to it.

Example 15.88: K-Nearest Neighbors search with a filter

```
float[] coverImageEmbeddingsVector = /*...*/
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.knn( 5 ).field( "coverImageEmbeddings" ).matching( coverImageEmbeddingsVector )
                .filter( f.match().field( "authors.firstName" ).matching( "isaac" ) ) )
        .fetchHits( 20 );
```

Combining `knn` with other predicates

A `knn` predicate can be combined with the regular text-search predicates. It can improve the quality of search results by increasing the score of documents that are more relevant based on vector embeddings characteristics:

Example 15.89: Enriching regular text search with K-Nearest Neighbors search

```
float[] coverImageEmbeddingsVector = /*...*/
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.bool()
                .must( f.match().field( "genre" ).matching( Genre.SCIENCE_FICTION ) ) ①
                .should( f.knn( 10 ).field( "coverImageEmbeddings" ).matching( coverImageEmbeddingsVector ) ) ②
        )
        .fetchHits( 20 );
```

① Find science fiction books.
② Improve the score of science fiction books that have a cover similar to the one we are searching for.

Filtering out irrelevant results with knn similarity

By its nature a knn predicate will always try to find k nearest vectors, even if the found vectors are quite far away from each other, i.e. are not that similar. This may lead to getting irrelevant results returned by the query.

To address this, a knn predicate allows configuring the minimum required similarity. If configured, knn predicate will find k nearest vectors and filter out any that will have similarity lower than this configured threshold. Note that the expected value for this property is a distance value between two vectors according to configured vector similarity.

Example 15.90: Filtering out irrelevant results

```
float[] coverImageEmbeddingsVector = /*...*/
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.knn( 5 ).field( "coverImageEmbeddings" ).matching( coverImageEmbeddingsVector ) ①
                .requiredMinimumSimilarity( 5 ) ) ②
        .fetchHits( 20 );
```

① Create a knn predicate as usual.
② Specify the required minimum similarity value, to filter out irrelevant results.

Alternatively, since in case of the knn predicate score and similarity are tight together as explained in this table, it maybe be sometimes simpler to apply a score based filter instead.

Example 15.91: Filtering out irrelevant results with score

```
float[] coverImageEmbeddingsVector = /*...*/
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.knn( 5 ).field( "coverImageEmbeddings" ).matching( coverImageEmbeddingsVector ) ①
                .requiredMinimumScore( 0.5f ) ) ②
        .fetchHits( 20 );
```

① Create a knn predicate as usual.
② Specify the required minimum score value, to filter out irrelevant results.

This configuration has a slightly different behavior with the OpenSearch distribution of an Elasticsearch backend. This distribution allows to use only one of the following options at the same time: k, minimum score, minimum similarity. Hence, if either `requiredMinimumSimilarity(..)` or `requiredMinimumScore(..)` is applied, then k value will be ignored and will not be sent in the request to the OpenSearch cluster.

Backend specifics and limitations

The parameter k has different behavior between backends, and their distributions.

With the Lucene backend k is the number that will limit the final amount of documents matched by a knn predicate. When using the Elastic distribution of the Elasticsearch backend k will be treated as both k and `num_candidates`. See the Elasticsearch documentation for more details. While when an OpenSearch distribution is used, k will be mapped to k attribute of knn query. Note that in this case you may get more than k results, when the index is configured to have more than one shard. See this section of the OpenSearch documentation for more details.

Using the `knn` predicate inside a nested predicate with the Elasticsearch backend has some limitations. In particular, when a tenant or routing filters are implicitly applied, the produced results may contain fewer documents than expected. To address this limitation, a schema change is required and should be addressed in one of the future major releases (HSEARCH-5085).

Other options

- The score of a `knn` predicate is variable by default (higher for "nearer" documents), but can be made constant with `.constantScore()`.
- The score of a `knn` predicate can be boosted for the whole predicate with a call to `.boost(…)` after `.matching(…)`.

15.2.20. `queryString`: match a user-provided query string

The `queryString` predicate matches documents according to a structured query given as a string. It allows building more advanced query strings and has more configuration options than a `simpleQueryString` predicate.

We will not go much into details of a query syntax in this guide To familiarize yourself with it, please refer to your backend (Elasticsearch /OpenSearch/Lucene) guides.

Example 15.92: Matching a query string

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.queryString().field( "description" )
                .matching(
                        "robots +(crime investigation disappearance)^10 +\"investigation help\"~2
-/(dis)?a[p]+ea?ance/" ) ) ①
        .fetchHits( 20 );
```

① This query string will result in a boolean query with 4 clauses:
- a should clause matching `robots`;
- two must clauses
  - another boolean query constructed from `(crime || investigation || disappearance)` string with a boost of `10`
  - a query matching the phrase `investigation help` with the phrase slop equals to `2`
- a must not clause matching a regular expression `(dis)?a[p]+ea?ance`

Note that each of the mentioned clauses may itself end up being translated into other types of queries.

Default boolean operator

By default, the query uses the OR operator if the operator is not explicitly defined. If you prefer using the AND operator as default, you can call `.defaultOperator(…)`.

Example 15.93: Matching a query string: AND as default operator

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.queryString().field( "description" )
                .matching( "robots investigation" )
                .defaultOperator( BooleanOperator.AND ) )
        .fetchHits( 20 );
```

Phrase slop

The phrase slop option defines how permissive a constructed phrase predicate will be; in other words, how many transpositions in the phrase are allowed for it to still be considered a match. With query predicate, this option can be set in the query string itself.

Example 15.94: Matching a query string: phrase slop as part of query string itself

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.queryString().field( "title" )
                .matching( "\"dawn robot\"~3" ) )
        .fetchHits( 20 );
```

Alternatively, `.phraseSlop(…)` can be applied to a query string predicate.

Example 15.95: Matching a query string: phrase slop as predicate option

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.queryString().field( "title" )
                .matching( "\"dawn robot\"" )
                .phraseSlop( 3 ) )
        .fetchHits( 20 );
```

Note that passing the value to `.phraseSlop(…)` sets the default phrase slop value, that can be overridden in the query string.

Example 15.96: Matching a query string: phrase slop as predicate option overridden by query

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.queryString().field( "title" )
                .matching( "\"dawn robot\"~3 -\"automatic detective\"" ) ①
                .phraseSlop( 1 ) ) ②
        .fetchHits( 20 );
```

① Query string is combination of two phrase queries. First phrase query `"dawn robot"` applies an override of the default phrase slop parameter and sets it to `3`. Second phrase query `"automatic detective"` uses the default phrase slop set in (2).

② Set the default pharse slop value to apply to the phrase queries that do not specify the slop explicitly in the query string.

Allowing leading wildcards

A query string can use a wildcard at any position within a query by default. If there is a need to prevent users from using leading wildcards, `.allowLeadingWildcard(..)` can be called with a `false` value to disallow such queries.

Example 15.97: Matching a query string: disallowing leading wildcards

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.queryString().field( "title" )
                .matching( "robo?" )
                .allowLeadingWildcard( false ) )
        .fetchHits( 20 );
```

In this example, a query uses the wildcard at the end of the word; hence, it is acceptable regardless of the option value. If queries like `?obot` or `*bot` were supplied, and at the same time, leading wildcards were disallowed it would have resulted in an exception being thrown.

Note that this option affects not just the whole query string, but individual clauses in that query string. For example, in a query string `robot ?etective` the wildcard `?` is not a leading character, but this query is broken down as two clauses for `robot` and `?etecitve` where in the second clause, the `?` wildcard becomes a leading character.

Enabling position increments

Position increments are enabled by default. Position increments are enabled by default, allowing phrase queries to take into account stop words removed by a `stopwords` filter. Position increments can be disabled as shown below, leading to a change in the behaviour of phrase queries: assuming that there is a phrase `book at the shelve` in the document, and the stop filter removes both `at` and `the`, with disabled position increments, phrase query `"book shelve"` will not match such document.

Example 15.98: Matching a query string: disabling position increments

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.queryString().field( "title" )
                .matching( "\"crime robots\"" )
                .enablePositionIncrements( false ) )
        .fetchHits( 20 );
```

Rewrite method

The rewrite method determines how the backend's query parser rewrites and scores multi-term queries.

To change the default `CONSTANT_SCORE` rewrite method, one of the allowed `RewriteMethod` enum values can be passed to `.rewriteMethod(RewriteMethod)`/`rewriteMethod(RewriteMethod, int)`.

Note, even though the default rewrite method is called `CONSTANT_SCORE` it does not mean that the matched documents' final score will be constant across all results, it is more about how the query parsing works internally. To achieve a constant score for results, see this documentation section on the query string predicate.

We will not go into details about different rewrite methods in this guide. To learn more about them, please refer to your backend's guide (Elasticsearch/OpenSearch/Lucene).

Example 15.99: Matching a query string: rewrite method

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.queryString().field( "title" )
                .matching(
                        // some complex query string
                )
                .rewriteMethod( RewriteMethod.CONSTANT_SCORE_BOOLEAN ) )
        .fetchHits( 20 );
```

`minimumShouldMatch`: fine-tuning how many `should` clauses are required to match

The resulting query parsed from the query string may result in a boolean query with should clauses. It may be helpful to control how many `should` clauses must match to consider a document as a match.

Example 15.100: Fine-tuning `should` clauses matching requirements with `minimumShouldMatch`

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.simpleQueryString().field( "title" )
                .matching( "crime robot investigate automatic detective" )
                .minimumShouldMatchNumber( 2 ) )
        .fetchHits( 20 );
```

This is similar to boolean or simple query string predicates `minimumShouldMatch` options.

Targeting multiple fields

Optionally, the predicate can target multiple fields. In that case, the predicate will match documents for which *any* of the given fields matches.

See Targeting multiple fields in one predicate.

If targeted fields have different analyzers, an exception will be thrown. You can avoid this by picking an analyzer explicitly, but make sure you know what you're doing.

Field types and expected format of field values

This predicate is applicable to most of the supported field types except `GeoPoint` and vector field types.

The format of string literals used in the query string is backend-specific. With the Lucene backend, the format of these literals should be compatible with the parsing logic defined in Property types with built-in value bridges, and for fields with custom bridges it must be defined. As for the Elasticsearch backend, see the Field types supported by the Elasticsearch backend.

Keep in mind that **not** all query constructs can be applied to non-string fields, e.g. creating regexp queries, using wildcards/slop/fuzziness will not work.

Other options

- The score of a `queryString` predicate is variable by default, but can be made constant with `.constantScore()`.

- The score of a `queryString` predicate can be boosted, either on a per-field basis with a call to `.boost(…)` just after `.field(…)`/`.fields(…)` or for the whole predicate with a call to `.boost(…)` after `.matching(…)`.
- The `queryString` predicate uses the search analyzer of targeted fields to analyze searched text by default, but this can be overridden.

15.2.21. `prefix`: match documents based on what a field starts with

The `prefix` predicate matches documents for which a given field has a value starting with a given string.

Example 15.101: Matching a simple prefix

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.prefix().field( "description" )
                .matching( "rob" ) )
        .fetchHits( 20 );
```

If a normalizer has been defined on the field, the prefix used in prefix predicates will be normalized.

If an analyzer has been defined on the field:

- when using the Elasticsearch backend, the prefixes won't be analyzed nor normalized, and will be expected to match a **single** indexed token. This may behave differently on an older versions of the underlying search engine Hence, please refer to the documentation of your particular backend version for the exact behaviour.
- when using the Lucene backend the prefixes will be normalized.

When the goal is to match user-provided query strings, the simple query string predicate should be preferred.

This predicate may also not be ideal for autocomplete purposes. Refer to your particular backend documentation to determine the best option to achieve autocomplete functionality.

Targeting multiple fields

Optionally, the predicate can target multiple fields. In that case, the predicate will match documents for which *any* of the given fields matches.

See Targeting multiple fields in one predicate.

15.2.22. `named`: call a predicate defined in the mapping

A `named` predicate, i.e. a predicate defined in the mapping, can be called and included in a query.

Below is an example that calls the named predicate from the example of section Defining named predicates.

Example 15.102: Calling a named predicate

```
List<ItemStock> hits = searchSession.search( ItemStock.class )
        .where( f -> f.named( "skuId.skuIdMatch" ) ①
                .param( "pattern", "*.WI2012" ) ) ②
        .fetchHits( 20 );
```

① The named predicate is referred to by its name, prefixed with the path of the object where the predicate was defined and a dot. Here, the predicate is named `skuIdMatch` and was defined on an object field named `skuId`. For named predicates defined at the root of the index, you can pass the predicate name directly, without any prefix.

② Named predicates can accept parameters, which will be handled as explained in Defining named predicates.

15.2.23. `withParameters`: create predicates accessing query parameters

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

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

You are encouraged to use incubating features so the development team can get feedback and improve them, but you

should be prepared to update code which relies on them as needed.

The `withParameters` predicate allows building predicates using query parameters. This predicate can be helpful when there is a need to execute a query with the same predicate but different input values, or when the same input value, passed as a query parameter, is used in multiple parts of a query, e.g. in a predicate, projection, sort, aggregation.

This type of predicate requires a function that accepts query parameters and returns a predicate. That function will get called at query building time.

Example 15.103: Creating a predicate with query parameters

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.withParameters( params -> f.bool() ①
                .should( f.match().field( "title" )
                        .matching( params.get( "title-param", String.class ) ) ) ②
                .filter( f.match().field( "genre" )
                        .matching( params.get( "genre-param", Genre.class ) ) ) ③
        ) )
        .param( "title-param", "robot" ) ④
        .param( "genre-param", Genre.CRIME_FICTION )
        .fetchHits( 20 );
```

① Start creating the `.withParameters()` predicate.
② Access the query parameter `title-param` of `String` type when constructing the predicate.
③ Access the query parameter `genre-param` of `Genre` enum type when constructing the predicate.
④ Set parameters required by the predicate at the query level.

15.2.24. Backend-specific extensions

By calling `.extension(…)` while building a query, it is possible to access backend-specific predicates.

As their name suggests, backend-specific predicates are not portable from one backend technology to the other.

Lucene: `fromLuceneQuery`

`.fromLuceneQuery(…)` turns a native Lucene `Query` into a Hibernate Search predicate.

This feature implies that application code rely on Lucene APIs directly.

An upgrade of Hibernate Search, even for a bugfix (micro) release, may require an upgrade of Lucene, which may lead to breaking API changes in Lucene.

If this happens, you will need to change application code to deal with the changes.

Example 15.104: Matching a native `org.apache.lucene.search.Query`

```
List<Book> hits = searchSession.search( Book.class )
        .extension( LuceneExtension.get() ) ①
        .where( f -> f.fromLuceneQuery( ②
                new RegexpQuery( new Term( "description", "neighbor|neighbour" ) )
        ) )
        .fetchHits( 20 );
```

① Build the query as usual, but using the Lucene extension so that Lucene-specific options are available.
② Add a predicate defined by a given Lucene `Query` object.

Elasticsearch: `fromJson`

`.fromJson(…)` turns JSON representing an Elasticsearch query into a Hibernate Search predicate.

This feature requires to directly manipulate JSON in application code.

> The syntax of this JSON may change:
>
> - when you upgrade the underlying Elasticsearch cluster to the next version;
> - when you upgrade Hibernate Search to the next version, even for a bugfix (micro) release.
>
> If this happens, you will need to change application code to deal with the changes.

Example 15.105: Matching a native Elasticsearch JSON query provided as a `JsonObject`

```
JsonObject jsonObject =
/* ... */; ①
List<Book> hits = searchSession.search( Book.class )
        .extension( ElasticsearchExtension.get() ) ②
        .where( f -> f.fromJson( jsonObject ) ) ③
        .fetchHits( 20 );
```

① Build a JSON object using Gson.
② Build the query as usual, but using the Lucene extension so that Lucene-specific options are available.
③ Add a predicate defined by a given `JsonObject`.

Example 15.106: Matching a native Elasticsearch JSON query provided as a JSON-formatted string

```
List<Book> hits = searchSession.search( Book.class )
        .extension( ElasticsearchExtension.get() ) ①
        .where( f -> f.fromJson( "{" ②
                + "    \"regexp\": {"
                + "        \"description\": \"neighbor|neighbour\""
                + "    }"
                + "}" ) )
        .fetchHits( 20 );
```

① Build the query as usual, but using the Lucene extension so that Lucene-specific options are available.
② Add a predicate defined by a given JSON-formatted string.

15.2.25. Options common to multiple predicate types

Targeting multiple fields in one predicate

Some predicates offer the ability to target multiple fields in the same predicate.

In that case, the predicate will match documents for which *any* of the given fields matches.

Below is an example with the `match` predicate.

Example 15.107: Matching a value in any of multiple fields

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.match()
                .field( "title" ).field( "description" )
                .matching( "robot" ) )
        .fetchHits( 20 );
```

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.match()
                .fields( "title", "description" )
                .matching( "robot" ) )
        .fetchHits( 20 );
```

It is possible to boost the score of each field separately; see Boosting the score of a predicate.

Tuning the score

Each predicate yields a score if it matched the document. The more relevant a document for a given predicate, the higher the score.

That score can be used when sorting by score (which is the default) to get more relevant hits at the top of the result list.

Below are a few ways to tune the score, and thus to get the most of the relevance sort.

Boosting the score of a predicate

The score of each predicate may be assigned a multiplier, called a *boost*:

- if a given predicate is more relevant to your search query than other predicates, assigning it a boost (multiplier) higher than 1 will increase its impact on the total document score.
- if a given predicate is less relevant to your search query than other predicates, assigning it a boost (multiplier) lower than 1 will decrease its impact on the total document score.

The boost should always be higher than 0.

Below is an example with the `match` predicate.

Example 15.108: Boosting on a per-predicate basis

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.and(
                f.match()
                        .field( "title" )
                        .matching( "robot" )
                        .boost( 2.0f ),
                f.match()
                        .field( "description" )
                        .matching( "self-aware" )
        ) )
        .fetchHits( 20 );
```

For predicates targeting multiple fields, it is also possible to assign more importance to matches on a given field (or set of fields) by calling `.boost(…)` after the call to `.field(…)`/`.fields(…)`.

Below is an example with the `match` predicate.

Example 15.109: Boosting on a per-field basis

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.match()
                .field( "title" ).boost( 2.0f )
                .field( "description" )
                .matching( "robot" ) )
        .fetchHits( 20 );
```

Variable and constant score

Some predicates already have a constant score by default, because a variable score doesn't make sense for them. For example, the `id` predicate has a constant score by default.

Predicates with a constant score have an "all-or-nothing" impact on the total document score: either a document matches and it will benefit from the score of this predicate (`1.0f` by default, but it can be boosted), or it doesn't match and it won't benefit from the score of this predicate.

Predicates with a variable score have a more subtle impact on the score. For example the `match` predicate, on a text field, will yield a higher score for documents that contain the term to match multiple times.

When this "variable score" behavior is not desired, you can suppress it by calling `.constantScore()`. This may be useful if only the fact that the predicate matched is relevant, but not the content of the document.

Below is an example with the `match` predicate.

Example 15.110: Making the score of a predicate constant

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.and(
                f.match()
                        .field( "genre" )
                        .matching( Genre.SCIENCE_FICTION )
                        .constantScore(),
                f.match()
                        .field( "title" )
                        .matching( "robot" )
                        .boost( 2.0f )
        ) )
        .fetchHits( 20 );
```

Alternatively, you can take advantage of the filter clause in boolean predicates, which is equivalent to a `must` clause but completely suppresses the impact of the clause on the score.

Overriding analysis

In some cases it might be necessary to use a different analyzer to analyze searched text than the one used to analyze indexed text.

This can be achieved by calling `.analyzer(…)` and passing the name of the analyzer to use.

Below is an example with the `match` predicate.

If you always apply the same analyzer when searching, you might want to configure a search analyzer on your field instead. Then you won't need to use `.analyzer(…)` when searching.

Example 15.111: Matching a value, analyzing it with a different analyzer

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.match()
                .field( "title_autocomplete" )
                .matching( "robo" )
                .analyzer( "autocomplete_query" ) )
        .fetchHits( 20 );
```

If you need to disable analysis of searched text completely, call `.skipAnalysis()`.

Example 15.112: Matching a value without analyzing it

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.match()
                .field( "title" )
                .matching( "robot" )
                .skipAnalysis() )
        .fetchHits( 20 );
```

15.3. Sort DSL

15.3.1. Basics

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

Example 15.113: Using custom sorts

```
SearchSession searchSession = /* ... */ ①
```

```
List<Book> result = searchSession.search( Book.class ) ②
        .where( f -> f.matchAll() )
        .sort( f -> f.field( "pageCount" ).desc() ③
                .then().field( "title_sort" ) )
        .fetchHits( 20 ); ④
```

① Retrieve the `SearchSession`.
② Start building the query as usual.
③ Mention that the results of the query are expected to be sorted on field "pageCount" in descending order, then (for those with the same page count) on field "title_sort" in ascending order. If the field does not exist or cannot be sorted on, an exception will be thrown.
④ Fetch the results, which will be sorted according to instructions.

Alternatively, if you don't want to use lambdas:

Example 15.114: Using custom sorts — object-based syntax

```
SearchSession searchSession = /* ... */

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

List<Book> result = searchSession.search( scope )
        .where( scope.predicate().matchAll().toPredicate() )
        .sort( scope.sort()
                .field( "pageCount" ).desc()
                .then().field( "title_sort" )
                .toSort() )
        .fetchHits( 20 );
```

In order to use sorts based on the value of a given field, you need to mark the field as sortable in the mapping.

This is not possible for full-text fields (multi-word text fields), in particular; see here for an explanation and some solutions.

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

15.3.2. `score`: sort by matching score (relevance)

`score` sorts on the score of each document:

- in descending order (the default), documents with a higher score appear first in the list of hits.
- in ascending order, documents with a lower score appear first in the list of hits.

The score is computed differently for each query, but roughly speaking you can consider that a higher score means that more predicates were matched, or they were matched better. Thus, the score of a given document is a representation of how relevant that document is to a particular query.

To get the most out of a sort by score, you will need to assign weight to your predicates by boosting some of them.

Advanced users may even want to change the scoring formula by specifying a different Similarity.

Sorting by score is the default, so it's generally not necessary to ask for a sort by score explicitly, but below is an example of how you can do it.

Example 15.115: Sorting by relevance

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.match().field( "title" )
                .matching( "robot dawn" ) )
        .sort( f -> f.score() )
        .fetchHits( 20 );
```

Options

- You can sort by ascending score by changing the sort order. However, this means the least relevant hits will appear first, which is completely pointless. This option is only provided for completeness.

15.3.3. `indexOrder`: sort according to the order of documents on storage

`indexOrder` sorts on the position of documents on internal storage.

This sort is not predictable, but is the most efficient. Use it when performance matters more than the order of hits.

Example 15.116: Sorting according to the order of documents on storage

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .sort( f -> f.indexOrder() )
        .fetchHits( 20 );
```

`indexOrder` should **not** be used to stabilize sorts. That's because the index order may change when the index is updated, or even without any change, after index segments are merged. See Stabilizing a sort for a solution to unstable sorts.

15.3.4. `field`: sort by field values

`field` sorts on the value of a given field for each document.

In order to use sorts based on the value of a given field, you need to mark the field as sortable in the mapping.

This is not possible for full-text fields (multi-word text fields), in particular; see here for an explanation and some solutions.

The values of GeoPoint fields cannot be compared directly and thus the `field` sort cannot be used on those fields.

Refer to the distance sort for these fields.

The sort order is defined as follows:

- in ascending order (the default), documents with a lower value appear first in the list of hits.
- in descending order, documents with a higher value appear first in the list of hits.

For text fields, "lower" means "lower in the alphabetical order".

Syntax

Example 15.117: Sorting by field values

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .sort( f -> f.field( "title_sort" ) )
        .fetchHits( 20 );
```

Options

- The sort order is ascending by default, but can be controlled explicitly with `.asc()`/`.desc()`.
- The behavior on missing values can be controlled explicitly with `.missing()`.
- The behavior on multivalued fields can be controlled explicitly with `.mode(…)`.
- For fields in nested objects, all nested objects are considered by default, but that can be controlled explicitly with `.filter(…)`.

15.3.5. `distance`: sort by distance to a point

`distance` sorts on the distance from a given center to the geo-point value of a given field for each document.

- in ascending order (the default), documents with a lower distance appear first in the list of hits.
- in descending order, documents with a higher distance appear first in the list of hits.

Prerequisites

In order for the `distance` sort to be available on a given field, you need to mark the field as sortable in the mapping.

Syntax

Example 15.118: Sorting by distance to a point

```
GeoPoint center = GeoPoint.of( 47.506060, 2.473916 );
List<Author> hits = searchSession.search( Author.class )
        .where( f -> f.matchAll() )
        .sort( f -> f.distance( "placeOfBirth", center ) )
        .fetchHits( 20 );
```

Options

- The sort order is ascending by default, but can be controlled explicitly with `.asc()`/`.desc()`.
- The behavior on multivalued fields can be controlled explicitly with `.mode(…)`.
- For fields in nested objects, all nested objects are considered by default, but that can be controlled explicitly with `.filter(…)`.

15.3.6. `withParameters`: create sorts using query parameters

> Features detailed below are *incubating*: they are still under active development.
>
> The usual compatibility policy does not apply: the contract of incubating elements (e.g. types, methods, configuration properties, etc.) may be altered in a backward-incompatible way — or even removed — in subsequent releases.
>
> You are encouraged to use incubating features so the development team can get feedback and improve them, but you should be prepared to update code which relies on them as needed.

The `withParameters` sort allows building sorts using query parameters.

This type of sort requires a function that accepts query parameters and returns a sort. That function will get called at query building time.

Example 15.119: Creating a sort with query parameters

```
GeoPoint center = GeoPoint.of( 47.506060, 2.473916 );
List<Author> hits = searchSession.search( Author.class )
        .where( f -> f.matchAll() )
        .sort( f -> f.withParameters( params -> f ①
                .distance( "placeOfBirth", params.get( "center", GeoPoint.class ) ) ) ) ②
        .param( "center", center ) ③
        .fetchHits( 20 );
```

① Start creating the `.withParameters()` sort.
② Access the query parameter `center` of `GeoPoint` type when constructing the sort.
③ Set parameters required by the sort at the query level.

15.3.7. `composite`: combine sorts

`composite` applies multiple sorts one after the other. It is useful when applying incomplete sorts.

Example 15.120: Sorting by multiple composed sorts using `composite()`

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .sort( f -> f.composite() ①
                .add( f.field( "genre_sort" ) ) ②
                .add( f.field( "title_sort" ) ) ) ③
        .fetchHits( 20 ); ④
```

① Start a `composite` sort.

② Add a `field` sort on the `genre_sort` field. Since many books share the same genre, this sort is incomplete: the relative order of books with the same genre is undetermined, and may change from one query execution to the other.

③ Add a `field` sort on the `title_sort` field. When two books have the same genre, their relative order will be determined by comparing their titles. If two books can have the same title, we can stabilize the sort even further by adding a last sort on the id.

④ The hits will be sorted by genre, then by title.

Alternatively, you can append a sort to another simply by calling `.then()` after the first sort:

Example 15.121: Sorting by multiple composed sorts using `then()`

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .sort( f -> f.field( "genre_sort" )
                .then().field( "title_sort" ) )
        .fetchHits( 20 );
```

Adding sorts dynamically with the lambda syntax

It is possible to define the `composite` sort inside a lambda expression. This is especially useful when inner sorts need to be added dynamically to the `composite` sort, for example based on user input.

Example 15.122: Easily composing sorts dynamically with the lambda syntax

```
MySearchParameters searchParameters = getSearchParameters(); ①
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .sort( f -> f.composite( b -> { ②
            for ( MySort mySort : searchParameters.getSorts() ) { ③
                switch ( mySort.getType() ) {
                    case GENRE:
                        b.add( f.field( "genre_sort" ).order( mySort.getOrder() ) );
                        break;
                    case TITLE:
                        b.add( f.field( "title_sort" ).order( mySort.getOrder() ) );
                        break;
                    case PAGE_COUNT:
                        b.add( f.field( "pageCount" ).order( mySort.getOrder() ) );
                        break;
                }
            }
        } ) )
        .fetchHits( 20 ); ④
```

① Get a custom object holding the search parameters provided by the user through a web form, for example.

② Call `.composite(Consumer)`. The consumer, implemented by a lambda expression, will receive a builder as an argument and will add sorts to that builder as necessary.

③ Inside the lambda, the code is free to do whatever is necessary before adding sorts. In this case, we iterate over user-selected sorts and add sorts accordingly.

④ The hits will be sorted according to sorts added by the lambda expression.

Stabilizing a sort

If your first sort (e.g. by field value) results in a tie for many documents (e.g. many documents have the same field value), you may want to append an arbitrary sort just to stabilize your sort: to make sure the search hits will always be in the same order if you execute the same query.

In most cases, a quick and easy solution for stabilizing a sort is to change your mapping to add a sortable field on your entity ID, and to append a `field` sort by id to your unstable sort:

Example 15.123: Stabilizing a sort

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .sort( f -> f.field( "genre_sort" ).then().field( "id_sort" ) )
        .fetchHits( 20 );
```

15.3.8. Backend-specific extensions

By calling `.extension(…)` while building a query, it is possible to access backend-specific sorts.

As their name suggests, backend-specific sorts are not portable from one backend technology to the other.

Lucene: `fromLuceneSort`

`.fromLuceneSort(…)` turns a native Lucene `Sort` into a Hibernate Search sort.

This feature implies that application code rely on Lucene APIs directly.

An upgrade of Hibernate Search, even for a bugfix (micro) release, may require an upgrade of Lucene, which may lead to breaking API changes in Lucene.

If this happens, you will need to change application code to deal with the changes.

Example 15.124: Sorting by a native `org.apache.lucene.search.Sort`

```
List<Book> hits = searchSession.search( Book.class )
        .extension( LuceneExtension.get() )
        .where( f -> f.matchAll() )
        .sort( f -> f.fromLuceneSort(
                new Sort(
                        new SortedSetSortField( "genre_sort", false ),
                        new SortedSetSortField( "title_sort", false )
                )
        ) )
        .fetchHits( 20 );
```

Lucene: `fromLuceneSortField`

`.fromLuceneSortField(…)` turns a native Lucene `SortField` into a Hibernate Search sort.

This feature implies that application code rely on Lucene APIs directly.

An upgrade of Hibernate Search, even for a bugfix (micro) release, may require an upgrade of Lucene, which may lead to breaking API changes in Lucene.

If this happens, you will need to change application code to deal with the changes.

Example 15.125: Sorting by a native `org.apache.lucene.search.SortField`

```
List<Book> hits = searchSession.search( Book.class )
        .extension( LuceneExtension.get() )
        .where( f -> f.matchAll() )
```

```
        .sort( f -> f.fromLuceneSortField(
                new SortedSetSortField( "title_sort", false )
        ) )
        .fetchHits( 20 );
```

Elasticsearch: `fromJson`

`.fromJson(…)` turns JSON representing an Elasticsearch sort into a Hibernate Search sort.

This feature requires to directly manipulate JSON in application code.

The syntax of this JSON may change:

- when you upgrade the underlying Elasticsearch cluster to the next version;
- when you upgrade Hibernate Search to the next version, even for a bugfix (micro) release.

If this happens, you will need to change application code to deal with the changes.

Example 15.126: Sorting by a native Elasticsearch JSON sort provided as a `JsonObject`

```
JsonObject jsonObject =
/* ... */;
List<Book> hits = searchSession.search( Book.class )
        .extension( ElasticsearchExtension.get() )
        .where( f -> f.matchAll() )
        .sort( f -> f.fromJson( jsonObject ) )
        .fetchHits( 20 );
```

Example 15.127: Sorting by a native Elasticsearch JSON sort provided as a JSON-formatted string

```
List<Book> hits = searchSession.search( Book.class )
        .extension( ElasticsearchExtension.get() )
        .where( f -> f.matchAll() )
        .sort( f -> f.fromJson( "{"
                + "     \"title_sort\": \"asc\""
                + "}" ) )
        .fetchHits( 20 );
```

15.3.9. Options common to multiple sort types

Sort order

Most sorts use the ascending order by default, with the notable exception of the score sort.

The order controlled explicitly through the following options:

- `.asc()` for an ascending order.
- `.desc()` for a descending order.
- `.order(…)` for an order defined by the given argument: `SortOrder.ASC`/`SortOrder.DESC`.

Below are a few examples with the field sort.

Example 15.128: Sorting by field values in explicitly ascending order with `asc()`

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .sort( f -> f.field( "title_sort" ).asc() )
        .fetchHits( 20 );
```

Example 15.129: Sorting by field values in explicitly descending order with `desc()`

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .sort( f -> f.field( "title_sort" ).desc() )
        .fetchHits( 20 );
```

Example 15.130: Sorting by field values in explicitly descending order with `order(…)`

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .sort( f -> f.field( "title_sort" ).order( SortOrder.DESC ) )
        .fetchHits( 20 );
```

Missing values

By default:

- For sorts by field values, the documents that do not have any value for a sort field will appear in the last position.
- For sorts by distance to a point, the documents that do not have any value for a sort field will be treated as if their distance from the given point was infinite.

The behavior for missing values can be controlled explicitly through the `.missing()` option:

- `.missing().first()` puts documents with no value in first position (regardless of the sort order).
- `.missing().last()` puts documents with no value in last position (regardless of the sort order).
- `.missing().lowest()` interprets missing values as the lowest value: it puts documents with no value in the first position when using ascending order or in the last position when using descending order.
- `.missing().highest()` interprets missing values as the highest value: it puts documents with no value in the last position when using ascending order or in the first position when using descending order.
- `.missing().use(…)` uses the given value as a default for documents with no value.

All these options are supported for sorts by field values and sorts by distance to a point using the Lucene backend.

When sorting by distance to a point using the Elasticsearch backend, due to limitations of the Elasticsearch APIs, only the following combinations are supported:

- `.missing().first()` using a descending order.
- `.missing().last()` using an ascending order.
- `.missing().highest()` using either an ascending or a descending order.

Below are a few examples with the field sort.

Example 15.131: Sorting by field values, documents with no value in first position

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .sort( f -> f.field( "pageCount" ).missing().first() )
        .fetchHits( 20 );
```

Example 15.132: Sorting by field values, documents with no value in last position

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .sort( f -> f.field( "pageCount" ).missing().last() )
        .fetchHits( 20 );
```

Example 15.133: Sorting by field values, documents with no value using a given default value

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .sort( f -> f.field( "pageCount" ).missing().use( 300 ) )
        .fetchHits( 20 );
```

Expected type of arguments

By default, `.use(…)` expects its argument to have the same type as the entity property corresponding to the target field.

For example, if an entity property is of type `java.util.Date`, the corresponding field may be of type `java.time.Instant`. `.use(…)` will expect its argument to be of type `java.util.Date` regardless.

This should generally be what you want, but if you ever need to bypass conversion and pass an unconverted argument (of type `java.time.Instant` in the example above) to `.use(…)`, see Type of arguments passed to the DSL.

Sort mode for multivalued fields

Documents that have multiple values for a sort field can be sorted too. A single value is picked for each document in order to compare it with order documents. How the value is picked is called the **sort mode**, specified using the `.mode(…)` option. The following sort modes are available:

Mode	Description	Supported value types	Unsupported value types
`SortMode.MIN`	Picks the lowest value for field sorts, the lowest distance for distance sorts. This is default for ascending sorts.	All.	-
`SortMode.MAX`	Picks the highest value for field sorts, the highest distance for distance sorts. This is default for descending sorts.	All.	-
`SortMode.SUM`	Computes the sum of all values for each document, and picks that sum for comparison with other documents.	Numeric fields (`long`, …).	Text and temporal fields (`String`, `LocalDate`, …), distance.
`SortMode.AVG`	Computes the arithmetic mean of all values for each document and picks that average for comparison with other documents.	Numeric and temporal fields (`long`, `LocalDate`, …), distance.	Text fields (`String`, …).
`SortMode.MEDIAN`	Computes the median of all values for each document, and picks that median for comparison with other documents.	Numeric and temporal fields (`long`, `LocalDate`, …), distance.	Text fields (`String`, …).

Below is an example with the field sort.

Example 15.134: Sorting by field values using the average value for each document

```
List<Author> hits = searchSession.search( Author.class )
        .where( f -> f.matchAll() )
        .sort( f -> f.field( "books.pageCount" ).mode( SortMode.AVG ) )
        .fetchHits( 20 );
```

Filter for fields in nested objects

When the sort field is located in a nested object, by default all nested objects will be considered for the sort and their values will be combined using the configured sort mode.

It is possible to filter the nested documents whose values will be considered for the sort using one of the `filter(…)` methods.

Below is an example with the field sort: authors are sorted by the average page count of their books, but only books of the "crime fiction" genre are considered:

Example 15.135: Sorting by field values using a filter for nested objects

```
List<Author> hits = searchSession.search( Author.class )
        .where( f -> f.matchAll() )
        .sort( f -> f.field( "books.pageCount" )
                .mode( SortMode.AVG )
                .filter( pf -> pf.match().field( "books.genre" )
                        .matching( Genre.CRIME_FICTION ) ) )
        .fetchHits( 20 );
```

15.4. Projection DSL

15.4.1. Basics

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

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

Example 15.136: Using projections to extract data from the index

```
SearchSession searchSession = /* ... */ ①

List<String> result = searchSession.search( Book.class ) ②
        .select( f -> f.field( "title", String.class ) ) ③
        .where( f -> f.matchAll() )
        .fetchHits( 20 ); ④
```

① Retrieve the `SearchSession`.
② Start building the query as usual.
③ Mention that the expected result of the query is a projection on field "title", of type String. If that type is not appropriate or if the field does not exist, an exception will be thrown.
④ Fetch the results, which will have the expected type.

Alternatively, if you don't want to use lambdas:

Example 15.137: Using projections to extract data from the index—object-based syntax

```
SearchSession searchSession = /* ... */

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

List<String> result = searchSession.search( scope )
        .select( scope.projection().field( "title", String.class )
                .toProjection() )
        .where( scope.predicate().matchAll().toPredicate() )
        .fetchHits( 20 );
```

In order to use projections based on the value of a given field, you need to mark the field as projectable in the mapping.

> This is optional with the Elasticsearch backend, where all fields are projectable by default.

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

15.4.2. Projecting to a custom (annotated) type

For more complex projections, it is possible to define a custom (annotated) record or class and have Hibernate Search infer the corresponding projections from the custom type's constructor parameters.

There are a few constraints to keep in mind when annotating a custom projection type:

- The custom projection type must be in the same JAR as entity types, or Hibernate Search will require additional configuration.
- When projecting on value fields or object fields, the path to the projected field is inferred from the constructor parameter name by default, but inference will fail if constructor parameter names are not included in the Java bytecode. Alternatively the path can be provided explicitly through `@FieldProjection(path = …)` /`@ObjectProjection(path = …)`, in which case Hibernate Search won't rely on constructor parameter names.
- When projecting on value fields, the constraints of the `field` projection still apply. In particular, with the Lucene backend, value fields involved in the projection must be configured as projectable.
- When projecting on object fields, the constraints of the `object` projection still apply. In particular, with the Lucene backend, multi-valued object fields involved in the projection must be configured as nested.

Example 15.138: Using a custom record type to project data from the index

```
@ProjectionConstructor ①
public record MyBookProjection(
        @IdProjection Integer id, ②
        String title, ③
        List<MyBookProjection.Author> authors) { ④
    @ProjectionConstructor ⑤
    public record Author(String firstName, String lastName) {
    }
}
```

① Annotate the record type with `@ProjectionConstructor`, either at the type level (if there's only one constructor) or at the constructor level (if there are multiple constructors).

② To project on the entity identifier, annotate the relevant constructor parameter with `@IdProjection`.
Most projections have a corresponding annotation that can be used on constructor parameters.

③ To project on a value field, add a constructor parameter named after that field and with the same type as that field. See Implicit inner projection inference for more information on how constructor parameters should be defined.
Alternatively, the field projection can be configured explicitly with `@FieldProjection`.

④ To project on an object field, add a constructor parameter named after that field and with its own custom projection type. Multivalued projections must be modeled as one of the multivalued containers available in `ProjectionCollector` or their supertype.
Alternatively, the object projection can be configured explicitly with `@ObjectProjection`.

⑤ Annotate any custom projection type used for object fields with `@ProjectionConstructor` as well.

```
List<MyBookProjection> hits = searchSession.search( Book.class )
        .select( MyBookProjection.class )①
        .where( f -> f.matchAll() )
        .fetchHits( 20 ); ②
```

① Pass the custom projection type to `.select(…)`. Hibernate Search will infer the inner projections from the custom type's constructor parameters.

② Each hit will be an instance of the custom projection type, populated with data retrieved from the index.

Custom, non-record classes can also be annotated with `@ProjectionConstructor`, which can be useful if you cannot use records for some reason (for example because you're still using Java 13 or below).

For more information about mapping custom projection types, see Mapping index content to custom types (projection constructors).

Beside `.select(Class<?>)`, some projections also allow using custom projection types; see the `composite` projection and the `object` projection. For more information about mapping projection types, see Mapping index content to custom types (projection constructors).

15.4.3. `documentReference`: return references to matched documents

The `documentReference` projection returns a reference to the matched document as a `DocumentReference` object.

Syntax

Example 15.139: Returning references to matched documents

```
List<DocumentReference> hits = searchSession.search( Book.class )
        .select( f -> f.documentReference() )
        .where( f -> f.matchAll() )
        .fetchHits( 20 );
```

Since it's a reference to the *document*, not the entity, `DocumentReference` only exposes low-level concepts such as the type name and the document identifier (a `String`). Use the `entityReference` projection to get a reference to the entity.

`@DocumentReferenceProjection` in projections to custom types

To achieve a `documentReference` projection inside a projection to an annotated custom type, use the `@DocumentReferenceProjection` annotation:

Example 15.140: Returning references to matched documents within a projection constructor

```
@ProjectionConstructor ①
public record MyBookDocRefAndTitleProjection(
        @DocumentReferenceProjection ②
        DocumentReference ref, ③
        String title ④
) {
}
```

① Annotate the record type with `@ProjectionConstructor`.
② Annotate the parameter that should receive the document reference with `@DocumentReferenceProjection`.
③ The type of the constructor parameter must be assignable from `DocumentReference`.
④ You can of course also declare other parameters with different projections; in this example an inferred projection to the `title` field.

```
List<MyBookDocRefAndTitleProjection> hits = searchSession.search( Book.class )
        .select( MyBookDocRefAndTitleProjection.class )①
        .where( f -> f.matchAll() )
        .fetchHits( 20 ); ②
```

① Pass the custom projection type to `.select(…)`.
② Each hit will be an instance of the custom projection type, populated with the requested document reference and field.

For programmatic mapping, use `DocumentReferenceProjectionBinder.create()`.

Example 15.141: Programmatic mapping of a `documentReference` projection within a projection constructor

```
TypeMappingStep myBookDocRefAndTitleProjection =
        mapping.type( MyBookDocRefAndTitleProjection.class );
myBookDocRefAndTitleProjection.mainConstructor()
        .projectionConstructor();
myBookDocRefAndTitleProjection.mainConstructor().parameter( 0 )
```

```
        .projection( DocumentReferenceProjectionBinder.create() );
```

15.4.4. `entityReference`: return references to matched entities

The `entityReference` projection returns a reference to the matched entity as an `EntityReference` object.

Syntax

Example 15.142: Returning references to matched entities

```
List<? extends EntityReference> hits = searchSession.search( Book.class )
        .select( f -> f.entityReference() )
        .where( f -> f.matchAll() )
        .fetchHits( 20 );
```

The entity does not get loaded as part of the projection. If you want the actual entity instance, use the `entity` projection

`@EntityReferenceProjection` in projections to custom types

To achieve an `entityReference` projection inside a projection to an annotated custom type, use the `@EntityReferenceProjection` annotation:

Example 15.143: Returning references to matched entities within a projection constructor

```
@ProjectionConstructor ①
public record MyBookEntityRefAndTitleProjection(
        @EntityReferenceProjection ②
        EntityReference ref, ③
        String title ④
) {
}
```

① Annotate the record type with `@ProjectionConstructor`.
② Annotate the parameter that should receive the entity reference with `@EntityReferenceProjection`.
③ The type of the constructor parameter must be assignable from `EntityReference`.
④ You can of course also declare other parameters with different projections; in this example an inferred projection to the `title` field.

```
List<MyBookEntityRefAndTitleProjection> hits = searchSession.search( Book.class )
        .select( MyBookEntityRefAndTitleProjection.class )①
        .where( f -> f.matchAll() )
        .fetchHits( 20 ); ②
```

① Pass the custom projection type to `.select(…)`.
② Each hit will be an instance of the custom projection type, populated with the requested entity reference and field.

For programmatic mapping, use `EntityReferenceProjectionBinder.create()`.

Example 15.144: Programmatic mapping of an `entityReference` projection within a projection constructor

```
TypeMappingStep myBookEntityRefAndTitleProjection =
        mapping.type( MyBookEntityRefAndTitleProjection.class );
myBookEntityRefAndTitleProjection.mainConstructor()
        .projectionConstructor();
myBookEntityRefAndTitleProjection.mainConstructor().parameter( 0 )
        .projection( EntityReferenceProjectionBinder.create() );
```

15.4.5. `id`: return identifiers of matched entities

The `id` projection returns the identifier of the matched entity.

Syntax

Example 15.145: Returning ids to matched entities, providing the identity type.

```
List<Integer> hits = searchSession.search( Book.class )
        .select( f -> f.id( Integer.class ) )
        .where( f -> f.matchAll() )
        .fetchHits( 20 );
```

If the provided identifier type does not match the type of identifiers for targeted entity types, an exception will be thrown. See also Type of projected values.

You can omit the "identifier type" argument, but then you will get projections of type `Object`:

Example 15.146: Returning ids to matched entities, without providing the identity type.

```
List<Object> hits = searchSession.search( Book.class )
        .select( f -> f.id() )
        .where( f -> f.matchAll() )
        .fetchHits( 20 );
```

`@IdProjection` in projections to custom types

To achieve an `id` projection inside a projection to an annotated custom type, use the `@IdProjection` annotation:

Example 15.147: Returning ids to matched entities within a projection constructor

```
@ProjectionConstructor ①
public record MyBookIdAndTitleProjection(
        @IdProjection ②
        Integer id, ③
        String title) { ④
}
```

① Annotate the record type with `@ProjectionConstructor`.
② Annotate the parameter that should receive the entity identifier with `@IdProjection`.
③ The type of the constructor parameter must be assignable from the type of the target entity's identifier.
④ You can of course also declare other parameters with different projections; in this example an inferred projection to the `title` field.

```
List<MyBookIdAndTitleProjection> hits = searchSession.search( Book.class )
        .select( MyBookIdAndTitleProjection.class )①
        .where( f -> f.matchAll() )
        .fetchHits( 20 ); ②
```

① Pass the custom projection type to `.select(…)`.
② Each hit will be an instance of the custom projection type, populated with the requested identifier and field.

For programmatic mapping, use `IdProjectionBinder.create()`.

Example 15.148: Programmatic mapping of an `id` projection within a projection constructor

```
TypeMappingStep myBookIdAndTitleProjectionMapping =
        mapping.type( MyBookIdAndTitleProjection.class );
myBookIdAndTitleProjectionMapping.mainConstructor()
        .projectionConstructor();
myBookIdAndTitleProjectionMapping.mainConstructor().parameter( 0 )
```

```
            .projection( IdProjectionBinder.create() );
```

15.4.6. `entity`: return matched entities

The `entity` projection returns the entity corresponding to the document that matched.

How the entity is loaded exactly depends on your mapper and configuration:

- With the Hibernate ORM integration, returned objects are managed entities loaded by Hibernate ORM from the database. You can use them as you would use any entity returned from traditional Hibernate ORM queries.
- With the Standalone POJO Mapper, entities are loaded from an external datastore if configured, or (failing that) are projected from the index if the entity type declares a projection constructor. If neither loading configuration nor projection constructor are found, the `entity` projection will simply fail.

Syntax

Example 15.149: Returning matched entities loaded from the database

```
List<Book> hits = searchSession.search( Book.class )
        .select( f -> f.entity() )
        .where( f -> f.matchAll() )
        .fetchHits( 20 );
```

If an entity cannot be loaded from the external datastore/database (e.g. it was deleted there, and the index wasn't updated yet), the hit will be omitted and won't appear in the returned `List` at all. The total hit count, however, will not take this omission into account.

Requesting a specific entity type

In some (rare) cases, the code creating the projection may have to work with a `SearchProjectionFactory<?, ?>`, i.e. a factory that carries no information regarding the type of loaded entities.

In those cases, it's possible to request a specific type of entities: Hibernate Search will check when the projection is created that the requested type matches the type of loaded entities.

Example 15.150: Requesting a specific entity type when for the `entity` projection

```
f.entity( Book.class )
```

`@EntityProjection` in projections to custom types

To achieve a `entity` projection inside a projection to an annotated custom type, use the `@EntityProjection` annotation:

Example 15.151: Returning matched entities loaded from the database within a projection constructor

```
@ProjectionConstructor ①
public record MyBookEntityAndTitleProjection(
        @EntityProjection ②
        Book entity, ③
        String title ④
) {
}
```

① Annotate the record type with `@ProjectionConstructor`.

② Annotate the parameter that should receive the entity with `@EntityProjection`.

③ The type of the constructor parameter must be assignable from the searched entity type — **all** of them in the case of queries targeting multiple indexes.

④ You can of course also declare other parameters with different projections; in this example an inferred projection to the `title` field.

```
List<MyBookEntityAndTitleProjection> hits = searchSession.search( Book.class )
        .select( MyBookEntityAndTitleProjection.class )①
        .where( f -> f.matchAll() )
        .fetchHits( 20 ); ②
```

① Pass the custom projection type to `.select(…)`.
② Each hit will be an instance of the custom projection type, populated with the requested entity and field.

For programmatic mapping, use `EntityProjectionBinder.create()`.

Example 15.152: Programmatic mapping of an `entity` projection within a projection constructor

```
TypeMappingStep myBookEntityAndTitleProjection =
        mapping.type( MyBookEntityAndTitleProjection.class );
myBookEntityAndTitleProjection.mainConstructor()
        .projectionConstructor();
myBookEntityAndTitleProjection.mainConstructor().parameter( 0 )
        .projection( EntityProjectionBinder.create() );
```

15.4.7. `field`: return field values from matched documents

The `field` projection returns the value of a given field for the matched document.

In order to use projections based on the value of a given field, you need to mark the field as projectable in the mapping.

This is optional with the Elasticsearch backend, where all fields are projectable by default.

Syntax

By default, the `field` projection returns a single value per document, so the code below will be enough for a single-valued field:

Example 15.153: Returning field values from matched documents

```
List<Genre> hits = searchSession.search( Book.class )
        .select( f -> f.field( "genre", Genre.class ) )
        .where( f -> f.matchAll() )
        .fetchHits( 20 );
```

Hibernate Search will throw an exception when building the query if you do this on a multivalued field. To project on multivalued fields, see Multivalued fields.

You can omit the "field type" argument, but then you will get projections of type `Object`:

Example 15.154: Returning field values from matched documents, without specifying the field type

```
List<Object> hits = searchSession.search( Book.class )
        .select( f -> f.field( "genre" ) )
        .where( f -> f.matchAll() )
        .fetchHits( 20 );
```

Multivalued fields

To return multiple values, and thus allow projection on multivalued fields, use `.collector(..)`. This will change the return type of the projection to `SomeContainer<T>` where `T` is what the single-valued projection would have returned.

Example 15.155: Returning field values from matched documents, for multivalued fields

```
List<List<String>> hits = searchSession.search( Book.class )
```

```
        .select( f -> f.field( "authors.lastName", String.class ).collector( ProjectionCollector.list() ) )
        .where( f -> f.matchAll() )
        .fetchHits( 20 );
```

Skipping conversion

By default, the values returned by the `field` projection have the same type as the entity property corresponding to the target field.

For example, if an entity property is of an enum type, the corresponding field may be of type `String`; the values returned by the `field` projection will be of the enum type regardless.

This should generally be what you want, but if you ever need to bypass conversion and have unconverted values returned to you instead (of type `String` in the example above), you can do it this way:

Example 15.156: Returning field values from matched documents, without converting the field value

```
List<String> hits = searchSession.search( Book.class )
        .select( f -> f.field( "genre", String.class, ValueModel.INDEX ) )
        .where( f -> f.matchAll() )
        .fetchHits( 20 );
```

See Type of projected values for more information.

`@FieldProjection` in projections to custom types

To achieve a `field` projection inside a projection to an annotated custom type, you can rely on the default inferred projection: when no annotations are present on a constructor parameter, it will be inferred to a field projection to the field with the same name as the constructor parameter (or an object projection, see here for details).

To force a field projection, or to customize the field projection further (for example to set the field path explicitly), use the `@FieldProjection` annotation on the constructor parameter:

Example 15.157: Returning field values from matched documents within a projection constructor

```
@ProjectionConstructor ①
public record MyBookTitleAndAuthorNamesProjection(
        @FieldProjection ②
        String title, ③
        @FieldProjection(path = "authors.lastName") ④
        List<String> authorLastNames ⑤
) {
}
```

① Annotate the record type with `@ProjectionConstructor`.
② Annotate the parameter that should receive the field value with `@FieldProjection`.
③ For single-valued projections, the type of the constructor parameter must be assignable from the type of the projected field.
④ Annotation attributes allow customization.
Here we're using a path that contains a dot, which we can't possibly include in the name of the Java constructor parameter.
⑤ For multi-valued projections, the type of the constructor parameter must be assignable from `List<T>`, where `T` is the type of the projected field or a supertype. See Inner projection and type for more information.

```
List<MyBookTitleAndAuthorNamesProjection> hits = searchSession.search( Book.class )
        .select( MyBookTitleAndAuthorNamesProjection.class )①
        .where( f -> f.matchAll() )
        .fetchHits( 20 ); ②
```

① Pass the custom projection type to `.select(…)`.
② Each hit will be an instance of the custom projection type, populated with the requested fields.

The annotation exposes the following attributes:

`path`

The path to the projected field.

If not set, it is inferred from the constructor parameter name.

Field path inference will fail if constructor parameter names are not included in the Java bytecode.

`convert`

How to convert values retrieved from the index.

For programmatic mapping, use `FieldProjectionBinder.create()`.

Example 15.158: Programmatic mapping of a `field` projection within a projection constructor

```
TypeMappingStep myBookTitleAndAuthorNamesProjectionMapping =
        mapping.type( MyBookTitleAndAuthorNamesProjection.class );
myBookTitleAndAuthorNamesProjectionMapping.mainConstructor()
        .projectionConstructor();
myBookTitleAndAuthorNamesProjectionMapping.mainConstructor().parameter( 0 )
        .projection( FieldProjectionBinder.create() );
myBookTitleAndAuthorNamesProjectionMapping.mainConstructor().parameter( 1 )
        .projection( FieldProjectionBinder.create( "authors.lastName" ) );
```

15.4.8. `score`: return the score of matched documents

The `score` projection returns the score of the matched document.

Syntax

Example 15.159: Returning the score of matched documents

```
List<Float> hits = searchSession.search( Book.class )
        .select( f -> f.score() )
        .where( f -> f.match().field( "title" )
                .matching( "robot dawn" ) )
        .fetchHits( 20 );
```

Two scores can only be reliably compared if they were computed during the very same query execution. Trying to compare scores from two separate query executions will only lead to confusing results, in particular if the predicates are different or if the content of the index changed enough to alter the frequency of some terms significantly.

On a related note, exposing scores to end users is generally not an easy task. See this article for some insight into what's wrong with displaying the score as a percentage, specifically.

`@ScoreProjection` in projections to custom types

To achieve a `score` projection inside a projection to an annotated custom type, use the `@ScoreProjection` annotation:

Example 15.160: Returning the score of matched documents within a projection constructor

```
@ProjectionConstructor ①
public record MyBookScoreAndTitleProjection(
        @ScoreProjection ②
        float score, ③
        String title ④
) {
}
```

① Annotate the record type with `@ProjectionConstructor`.
② Annotate the parameter that should receive the score with `@ScoreProjection`.

③ The type of the constructor parameter must be assignable from `float`.
④ You can of course also declare other parameters with different projections; in this example an inferred projection to the `title` field.

```
List<MyBookScoreAndTitleProjection> hits = searchSession.search( Book.class )
        .select( MyBookScoreAndTitleProjection.class )①
        .where( f -> f.matchAll() )
        .fetchHits( 20 ); ②
```

① Pass the custom projection type to `.select(…)`.
② Each hit will be an instance of the custom projection type, populated with the requested score and field.

For programmatic mapping, use `ScoreProjectionBinder.create()`.

Example 15.161: Programmatic mapping of a `score` projection within a projection constructor

```
TypeMappingStep myBookScoreAndTitleProjection =
        mapping.type( MyBookScoreAndTitleProjection.class );
myBookScoreAndTitleProjection.mainConstructor()
        .projectionConstructor();
myBookScoreAndTitleProjection.mainConstructor().parameter( 0 )
        .projection( ScoreProjectionBinder.create() );
```

15.4.9. `distance`: return the distance to a point

The `distance` projection returns the distance between a given point and the geo-point value of a given field for the matched document.

In order to use projections based on the value of a given field, you need to mark the field as projectable in the mapping.

This is optional with the Elasticsearch backend, where all fields are projectable by default.

Syntax

By default, the `distance` projection returns a single value per document, so the code below will be enough for a single-valued field:

Example 15.162: Returning the distance to a point

```
GeoPoint center = GeoPoint.of( 47.506060, 2.473916 );
SearchResult<Double> result = searchSession.search( Author.class )
        .select( f -> f.distance( "placeOfBirth", center ) )
        .where( f -> f.matchAll() )
        .fetch( 20 );
```

Hibernate Search will throw an exception when building the query if you do this on a multivalued field. To project on multivalued fields, see Multivalued fields.

The returned distance is in meters by default, but you can pick a different unit:

Example 15.163: Returning the distance to a point with a given distance unit

```
GeoPoint center = GeoPoint.of( 47.506060, 2.473916 );
SearchResult<Double> result = searchSession.search( Author.class )
        .select( f -> f.distance( "placeOfBirth", center )
                .unit( DistanceUnit.KILOMETERS ) )
        .where( f -> f.matchAll() )
        .fetch( 20 );
```

Multivalued fields

To return multiple values, and thus allow projection on multivalued fields, use `.collector(..)`. This will change the return type of the projection to `SomeContainer<Double>`.

Example 15.164: Returning the distance to a point, for multivalued fields

```
GeoPoint center = GeoPoint.of( 47.506060, 2.473916 );
SearchResult<List<Double>> result = searchSession.search( Book.class )
        .select( f -> f.distance( "authors.placeOfBirth", center ).collector( ProjectionCollector.list() ) )
        .where( f -> f.matchAll() )
        .fetch( 20 );
```

`@DistanceProjection` in projections to custom types

To achieve a `distance` projection inside a projection to an annotated custom type, use the `@DistanceProjection` annotation on the constructor parameter:

Example 15.165: Returning distance from a center point defined as a parameter to the field value in matched documents within a projection constructor

```
@ProjectionConstructor ①
public record MyAuthorPlaceProjection(
        @DistanceProjection( ②
                fromParam = "point-param", ③
                path = "placeOfBirth") ④
        Double distance ) { ⑤
}
```

① Annotate the record type with `@ProjectionConstructor`.
② Annotate the parameter that should receive the distance value with `@DistanceProjection`.
③ Specify the query parameter that will be used to calculate the distance from.
④ Optionally, customize the path, since most likely the `GeoPoint` property of the entity will have a different name from the distance property in a projection.
⑤ For single-valued projections, the type of the constructor parameter must be assignable from `Double`.
For multi-valued projections, the type of the constructor parameter must be assignable from `List<T>`, where `T` is `Double` or a supertype.

```
List<MyAuthorPlaceProjection> hits = searchSession.search( Author.class )
        .select( MyAuthorPlaceProjection.class )①
        .where( f -> f.matchAll() )
        .param( "point-param", GeoPoint.of( latitude, longitude ) ) ②
        .fetchHits( 20 ); ③
```

① Pass the custom projection type to `.select(…)`.
② Pass a query parameter value, with the same name `point-param` as in the `@DistanceProjection` `fromParam` of a projection constructor.
③ Each hit will be an instance of the custom projection type, populated with the requested fields.

The annotation exposes the following attributes:

`fromParam`

The name of a query parameter that will represent a point, from which the distance to the field value will be calculated.

This is a required attribute.

`path`

The path to the projected field.

If not set, it is inferred from the constructor parameter name.

Field path inference will fail if constructor parameter names are not included in the Java bytecode.

`unit`

: The unit of the computed distance (default is meters).

For programmatic mapping, use `DistanceProjectionBinder.create(..)`.

Example 15.166: Programmatic mapping of a `distance` projection within a projection constructor

```
TypeMappingStep myAuthorPlaceProjection =
        mapping.type( MyAuthorPlaceProjection.class );
myAuthorPlaceProjection.mainConstructor()
        .projectionConstructor();
myAuthorPlaceProjection.mainConstructor().parameter( 0 )
        .projection( DistanceProjectionBinder.create( "placeOfBirth", "point-param" ) );
```

15.4.10. `composite`: combine projections

Basics

The `composite` projection applies multiple projections and combines their results, either as a `List<?>` or as a single object generated using a custom transformer.

To preserve type-safety, you can provide a custom transformer. The transformer can be a `Function`, a `BiFunction`, or a `org.hibernate.search.util.common.function.TriFunction`, depending on the number of inner projections. It will receive values returned by inner projections and return an object combining these values.

Example 15.167: Returning custom objects created from multiple projected values with `.composite().from(…).as(…)`

```
List<MyPair<String, Genre>> hits = searchSession.search( Book.class )
        .select( f -> f.composite() ①
                .from( f.field( "title", String.class ), ②
                        f.field( "genre", Genre.class ) ) ③
                .as( MyPair::new ) )④
        .where( f -> f.matchAll() )
        .fetchHits( 20 ); ⑤
```

① Call `.composite()`.

② Define the first inner projection as a projection on the `title` field.

③ Define the second inner projection as a projection on the `genre` field.

④ Define the result of the composite projection as the result of calling the constructor of a custom object, `MyPair`. The constructor of `MyPair` will be called for each matched document, with the value of the `title` field as its first argument, and the value of the `genre` field as its second argument.

⑤ Each hit will be an instance of `MyPair`. Thus, the list of hits will be an instance of `List<MyPair>`.

Composing more than 3 inner projections

For complex projections, consider projecting to a custom (annotated) type.

If you pass more than 3 projections as arguments to `from(…)`, then the transform function will have to take a `List<?>` as an argument, and will be set using `asList(…)` instead of `as(...)`:

Example 15.168: Returning custom objects created from multiple projected values with `.composite().from(…).asList(…)`

```
List<MyTuple4<String, Genre, Integer, String>> hits = searchSession.search( Book.class )
        .select( f -> f.composite() ①
                .from( f.field( "title", String.class ), ②
                        f.field( "genre", Genre.class ), ③
                        f.field( "pageCount", Integer.class ), ④
                        f.field( "description", String.class ) ) ⑤
                .asList( list -> ⑥
                    new MyTuple4<>( (String) list.get( 0 ), (Genre) list.get( 1 ),
                            (Integer) list.get( 2 ), (String) list.get( 3 ) ) ) )
```

```
            .where( f -> f.matchAll() )
            .fetchHits( 20 ); ⑦
```

① Call `.composite()`.
② Define the first inner projection as a projection on the `title` field.
③ Define the second inner projection as a projection on the `genre` field.
④ Define the third inner projection as a projection on the `pageCount` field.
⑤ Define the fourth inner projection as a projection on the `description` field.
⑥ Define the result of the object projection as the result of calling a lambda. The lambda will take elements of the list (the results of projections defined above, in order), cast them, and pass them to the constructor of a custom class, `MyTuple4`.
⑦ Each hit will be an instance of `MyTuple4`. Thus, the list of hits will be an instance of `List<MyTuple4>`.

Projecting to a `List<?>` or `Object[]`

If you don't mind receiving the result of inner projections as a `List<?>`, you can do without the transformer by calling `asList()`:

Example 15.169: Returning a `List` of projected values with `.composite().add(…).asList()`

```
List<List<?>> hits = searchSession.search( Book.class )
        .select( f -> f.composite() ①
                .from( f.field( "title", String.class ), ②
                        f.field( "genre", Genre.class ) ) ③
                .asList() ) ④
        .where( f -> f.matchAll() )
        .fetchHits( 20 ); ⑤
```

① Call `.composite()`.
② Define the first inner projection as a projection on the `title` field.
③ Define the second inner projection as a projection on the `genre` field.
④ Define the result of the projection as a list, meaning the hits will be `List` instances with the value of the `title` field of the matched document at index `0`, and the value of the `genre` field of the matched document at index `1`.
⑤ Each hit will be an instance of `List<?>`: a list containing the result of the inner projections, in the given order. Thus, the list of hits will be an instance of `List<List<?>>`.

Similarly, to get the result of inner projections as an array (`Object[]`), you can do without the transformer by calling `asArray()`:

Example 15.170: Returning an array of projected values with `.composite(…).add(…).asArray()`

```
List<Object[]> hits = searchSession.search( Book.class )
        .select( f -> f.composite() ①
                .from( f.field( "title", String.class ), ②
                        f.field( "genre", Genre.class ) ) ③
                .asArray() ) ④
        .where( f -> f.matchAll() )
        .fetchHits( 20 ); ⑤
```

① Call `.composite()`.
② Define the first inner projection as a projection on the `title` field.
③ Define the second inner projection as a projection on the `genre` field.
④ Define the result of the projection as an array, meaning the hits will be `Object[]` instances with the value of the `title` field of the matched document at index `0`, and the value of the `genre` field of the matched document at index `1`.
⑤ Each hit will be an instance of `Object[]`: a array of objects containing the result of the inner projections, in the given order. Thus, the list of hits will be an instance of `List<Object[]>`.

Alternatively, to get the result as a `List<?>`, you can use the shorter variant of `.composite(…)` that directly takes projections as arguments:

Example 15.171: Returning a `List` of projected values with `.composite(…)`

```
List<List<?>> hits = searchSession.search( Book.class )
        .select( f -> f.composite( ①
```

```
                f.field( "title", String.class ), ②
                f.field( "genre", Genre.class ) ③
        ) )
        .where( f -> f.matchAll() )
        .fetchHits( 20 ); ④
```

① Call `.composite(…)`.

② Define the first projection to combine as a projection on the `title` field.

③ Define the second projection to combine as a projection on the `genre` field.

④ Each hit will be an instance of `List<?>`: a list containing the result of the inner projections, in the given order. Thus, the list of hits will be an instance of `List<List<?>>`.

Projecting to a custom (annotated) type

For more complex composite projections, it is possible to define a custom (annotated) record or class and have Hibernate Search infer the corresponding inner projections from the custom type's constructor parameters. This is similar to the projection to a custom (annotated) type through `.select(…)`.

There are a few constraints to keep in mind when annotating a custom projection type:

- The custom projection type must be in the same JAR as entity types, or Hibernate Search will require additional configuration.
- When projecting on value fields or object fields, the path to the projected field is inferred from the constructor parameter name by default, but inference will fail if constructor parameter names are not included in the Java bytecode. Alternatively the path can be provided explicitly through `@FieldProjection(path = …)` /`@ObjectProjection(path = …)`, in which case Hibernate Search won't rely on constructor parameter names.
- When projecting on value fields, the constraints of the `field` projection still apply. In particular, with the Lucene backend, value fields involved in the projection must be configured as projectable.
- When projecting on object fields, the constraints of the `object` projection still apply. In particular, with the Lucene backend, multi-valued object fields involved in the projection must be configured as nested.

Example 15.172: Using a custom record type to project data from the index

```
@ProjectionConstructor ①
public record MyBookProjection(
        @IdProjection Integer id, ②
        String title, ③
        List<MyBookProjection.Author> authors) { ④
    @ProjectionConstructor ⑤
    public record Author(String firstName, String lastName) {
    }
}
```

① Annotate the record type with `@ProjectionConstructor`, either at the type level (if there's only one constructor) or at the constructor level (if there are multiple constructors).

② To project on the entity identifier, annotate the relevant constructor parameter with `@IdProjection`.

Most projections have a corresponding annotation that can be used on constructor parameters.

③ To project on a value field, add a constructor parameter named after that field and with the same type as that field. See Implicit inner projection inference for more information on how constructor parameters should be defined.

Alternatively, the field projection can be configured explicitly with `@FieldProjection`.

④ To project on an object field, add a constructor parameter named after that field and with its own custom projection type. Multivalued projections must be modeled as one of the multivalued containers for which a collector is available in `ProjectionCollector` or their supertype.

Alternatively, the object projection can be configured explicitly with `@ObjectProjection`.

⑤ Annotate any custom projection type used for object fields with `@ProjectionConstructor` as well.

```
List<MyBookProjection> hits = searchSession.search( Book.class )
        .select( f -> f.composite() ①
                .as( MyBookProjection.class ) )②
        .where( f -> f.matchAll() )
        .fetchHits( 20 ); ③
```

① Call `.composite()`.
② Define the result of the projection as a custom (annotated) type. Hibernate Search will infer the inner projections from the custom type's constructor parameters.
③ Each hit will be an instance of the custom projection type, populated with data retrieved from the index.

Custom, non-record classes can also be annotated with `@ProjectionConstructor`, which can be useful if you cannot use records for some reason (for example because you're still using Java 13 or below).

For more information about mapping custom projection types, see Mapping index content to custom types (projection constructors).

`@CompositeProjection` in projections to custom types

To achieve a `composite` projection inside a projection to an annotated custom type, use the `@CompositeProjection` annotation on the constructor parameter:

Example 15.173: Returning custom objects created from multiple projections within a projection constructor

```
@ProjectionConstructor ①
public record MyBookMiscInfoAndTitleProjection(
        @CompositeProjection ②
        MiscInfo miscInfo, ③
        String title ④
) {

    @ProjectionConstructor ③
    public record MiscInfo(
            Genre genre,
            Integer pageCount
    ) {
    }
}
```

① Annotate the record type with `@ProjectionConstructor`.
② Annotate the parameter that should receive the composite projection with `@CompositeProjection`.
③ The type of the constructor parameter must have a projection constructor itself. See Inner projection and type for more information.
④ You can of course also declare other parameters with different projections; in this example an inferred projection to the `title` field.

```
List<MyBookMiscInfoAndTitleProjection> hits = searchSession.search( Book.class )
        .select( MyBookMiscInfoAndTitleProjection.class )①
        .where( f -> f.matchAll() )
        .fetchHits( 20 ); ②
```

① Pass the custom projection type to `.select(…)`.
② Each hit will be an instance of the custom projection type, populated with the requested composite projection.

For programmatic mapping, use `CompositeProjectionBinder.create()`.

Example 15.174: Programmatic mapping of a `composite` projection within a projection constructor

```
TypeMappingStep myBookMiscInfoAndTitleProjection =
        mapping.type( MyBookMiscInfoAndTitleProjection.class );
myBookMiscInfoAndTitleProjection.mainConstructor()
        .projectionConstructor();
myBookMiscInfoAndTitleProjection.mainConstructor().parameter( 0 )
        .projection( CompositeProjectionBinder.create() );
TypeMappingStep miscInfoProjection =
        mapping.type( MyBookMiscInfoAndTitleProjection.MiscInfo.class );
miscInfoProjection.mainConstructor().projectionConstructor();
```

Deprecated variants

Features detailed in this section are *deprecated*: they should be avoided in favor of non-deprecated alternatives.

The usual compatibility policy applies, meaning the features are expected to remain available at least until the next major version of Hibernate Search. Beyond that, they may be altered in a backward-incompatible way—or even removed.

Usage of deprecated features is not recommended.

A few `.composite(…)` methods accepting both a function and a list of projections are available on `SearchProjectionFactory`, but they are deprecated.

Example 15.175: Deprecated variant of `composite`

```
List<MyPair<String, Genre>> hits = searchSession.search( Book.class )
        .select( f -> f.composite( ①
                MyPair::new, ②
                f.field( "title", String.class ), ③
                f.field( "genre", Genre.class ) ④
        ) )
        .where( f -> f.matchAll() )
        .fetchHits( 20 ); ⑤
```

① Call `.composite(…)`.

② Define the transformer as the constructor of a custom object, `MyPair`.

③ Define the first projection to combine as a projection on the `title` field, meaning the constructor of `MyPair` will be called for each matched document with the value of the `title` field as its first argument.

④ Define the second projection to combine as a projection on the `genre` field, meaning the constructor of `MyPair` will be called for each matched document with the value of the `genre` field as its second argument.

⑤ Each hit will be an instance of `MyPair`. Thus, the list of hits will be an instance of `List<MyPair>`.

15.4.11. `object`: return one value per object in an object field

The `object` projection yields one projected value for each object in a given object field, the value being generated by applying multiple inner projections and combining their results either as a `List<?>` or as a single object generated using a custom transformer.

The `object` projection may seem very similar to the `composite` projection, and its definition via the Search DSL certainly is indeed similar.

However, there are two key differences:

1. The `object` projection will yield `null` when projecting on a single-valued object field if the object was null when indexing.
2. The `object` projection will yield multiple values when projecting on a multivalued object field if there were multiple objects when indexing.

With the Lucene backend, the object projection has a few limitations:

1. It is only available for single-valued object fields regardless of their structure, or multi-valued object fields with a `NESTED` structure.
2. It will never yield `null` objects for multi-valued object fields. The Lucene backend does not index `null` objects, and thus cannot find them when searching.

These limitations do not apply to the Elasticsearch backend.

Syntax

To preserve type-safety, you can provide a custom transformer. The transformer can be a `Function`, a `BiFunction`, or a `org.hibernate.search.util.common.function.TriFunction`, depending on the number of inner projections. It will receive values returned by inner projections and return an object combining these values.

Example 15.176: Returning custom objects created from an object field with `.object(…).from(…).as(…)`

```
List<List<MyAuthorName>> hits = searchSession.search( Book.class )
        .select( f -> f.object( "authors" ) ①
                .from( f.field( "authors.firstName", String.class ), ②
                        f.field( "authors.lastName", String.class ) ) ③
                .as( MyAuthorName::new ) ④
                .collector( ProjectionCollector.list() ) ) ⑤
        .where( f -> f.matchAll() )
        .fetchHits( 20 ); ⑥
```

① Call `.object( "authors" )`.

② Define the first inner projection as a projection on the `firstName` field of `authors`.

③ Define the second inner projection as a projection on the `lastName` field of `authors`.

④ Define the result of the object projection as the result of calling the constructor of a custom object, `MyAuthorName`. The constructor of `MyAuthorName` will be called for each object in the `authors` object field, with the value of the `authors.firstName` field as its first argument, and the value of the `authors.lastName` field as its second argument.

⑤ Define the projection as multivalued, meaning it will yield values of type `List<MyAuthorName>`: one `MyAuthorName` per object in the `authors` object field.

⑥ Each hit will be an instance of `List<MyAuthorName>`. Thus, the list of hits will be an instance of `List<List<MyAuthorName>>`.

Composing more than 3 inner projections

For complex projections, consider projecting to a custom (annotated) type.

If you pass more than 3 projections as arguments, then the transform function will have to take a `List<?>` as an argument, and will be set using `asList(…)` instead of `as(…)`:

Example 15.177: Returning custom objects created from an object field with `.object(…).from(…).asList(…)`

```
GeoPoint center = GeoPoint.of( 53.970000, 32.150000 );
List<List<MyAuthorNameAndBirthDateAndPlaceOfBirthDistance>> hits = searchSession
        .search( Book.class )
        .select( f -> f.object( "authors" ) ①
                .from( f.field( "authors.firstName", String.class ), ②
                        f.field( "authors.lastName", String.class ), ③
                        f.field( "authors.birthDate", LocalDate.class ), ④
                        f.distance( "authors.placeOfBirth", center ) ⑤
                                .unit( DistanceUnit.KILOMETERS ) )
                .asList( list -> ⑥
                        new MyAuthorNameAndBirthDateAndPlaceOfBirthDistance(
                                (String) list.get( 0 ), (String) list.get( 1 ),
                                (LocalDate) list.get( 2 ), (Double) list.get( 3 ) ) )
                .collector( ProjectionCollector.list() ) ) ⑦
        .where( f -> f.matchAll() )
        .fetchHits( 20 ); ⑧
```

① Call `.object( "authors" )`.

② Define the first inner projection as a projection on the `firstName` field of `authors`.

③ Define the second inner projection as a projection on the `lastName` field of `authors`.

④ Define the third inner projection as a projection on the `birthDate` field of `authors`.

⑤ Define the fourth inner projection as a distance projection on the `placeOfBirth` field with the given center and unit.

⑥ Define the result of the object projection as the result of calling a lambda. The lambda will take elements of the list (the results of projections defined above, in order), cast them, and pass them to the constructor of a custom class, `MyAuthorNameAndBirthDateAndPlaceOfBirthDistance`.

⑦ Define the projection as multivalued, meaning it will yield values of type `List<MyAuthorNameAndBirthDateAndPlaceOfBirthDistance>`: one `MyAuthorNameAndBirthDateAndPlaceOfBirthDistance` per object in the `authors` object field. instead of just `MyAuthorNameAndBirthDateAndPlaceOfBirthDistance`.

⑧ Each hit will be an instance of `List<MyAuthorNameAndBirthDateAndPlaceOfBirthDistance>`. Thus, the list of hits will be an instance of `List<List<MyAuthorNameAndBirthDateAndPlaceOfBirthDistance>>`.

Similarly, `asArray(…)` can be used to get passed an `Object[]` argument instead of `List<?>`.

Example 15.178: Returning custom objects created from an object field with `.object(…).from(…).asArray(…)`

```
GeoPoint center = GeoPoint.of( 53.970000, 32.150000 );
List<List<MyAuthorNameAndBirthDateAndPlaceOfBirthDistance>> hits = searchSession
        .search( Book.class )
        .select( f -> f.object( "authors" ) ①
                .from( f.field( "authors.firstName", String.class ), ②
                        f.field( "authors.lastName", String.class ), ③
                        f.field( "authors.birthDate", LocalDate.class ), ④
                        f.distance( "authors.placeOfBirth", center ) ⑤
                                .unit( DistanceUnit.KILOMETERS ) )
                .asArray( array -> ⑥
                        new MyAuthorNameAndBirthDateAndPlaceOfBirthDistance(
                                (String) array[0], (String) array[1],
                                (LocalDate) array[2], (Double) array[3] ) )
                .collector( ProjectionCollector.list() ) ) ⑦
        .where( f -> f.matchAll() )
        .fetchHits( 20 ); ⑧
```

① Call `.object( "authors" )`.

② Define the first inner projection as a projection on the `firstName` field of `authors`.

③ Define the second inner projection as a projection on the `lastName` field of `authors`.

④ Define the third inner projection as a projection on the `birthDate` field of `authors`.

⑤ Define the fourth inner projection as a distance projection on the `placeOfBirth` field with the given center and unit.

⑥ Define the result of the object projection as the result of calling a lambda. The lambda will take elements of the array (the results of projections defined above, in order), cast them, and pass them to the constructor of a custom class, `MyAuthorNameAndBirthDateAndPlaceOfBirthDistance`.

⑦ Define the projection as multivalued, meaning it will yield values of type `List<MyAuthorNameAndBirthDateAndPlaceOfBirthDistance>`: one `MyAuthorNameAndBirthDateAndPlaceOfBirthDistance` per object in the `authors` object field. instead of just `MyAuthorNameAndBirthDateAndPlaceOfBirthDistance`.

⑧ Each hit will be an instance of `List<MyAuthorNameAndBirthDateAndPlaceOfBirthDistance>`. Thus, the list of hits will be an instance of `List<List<MyAuthorNameAndBirthDateAndPlaceOfBirthDistance>>`.

Projecting to a `List<?>` or `Object[]`

If you don't mind receiving the result of inner projections as a `List<?>`, you can do without the transformer by calling `asList()`:

Example 15.179: Returning a `List` of projected values with `.object(…).add(…).asList()`

```
List<List<List<?>>> hits = searchSession.search( Book.class )
        .select( f -> f.object( "authors" ) ①
                .from( f.field( "authors.firstName", String.class ), ②
                        f.field( "authors.lastName", String.class ) ) ③
                .asList() ④
                .collector( ProjectionCollector.list() ) ) ⑤
        .where( f -> f.matchAll() )
        .fetchHits( 20 ); ⑥
```

① Call `.object( "authors" )`.

② Define the first inner projection as a projection on the `firstName` field of `authors`.

③ Define the second inner projection as a projection on the `lastName` field of `authors`.

④ Define the result of the projection as a list, meaning the hits will be `List` instances with at index `0` the value of the `firstName` field of `authors` , and at index `1` the value of the `lastName` field of `authors`.

⑤ Define the projection as multivalued, meaning it will yield values of type `List<List<?>>`: one `List<?>` per object in the `authors` object field.

⑥ Each hit will be an instance of `List<List<?>>`: a list containing one list per author, which in turns contains the result of the inner projections, in the given order. Thus, the list of hits will be an instance of `List<List<List<?>>>`.

Similarly, to get the result of inner projections as an array (`Object[]`), you can do without the transformer by calling `asArray()`:

Example 15.180: Returning an array of projected values with `.object(…).add(…).asArray()`

```
List<List<Object[]>> hits = searchSession.search( Book.class )
        .select( f -> f.object( "authors" ) ①
                .from( f.field( "authors.firstName", String.class ), ②
                        f.field( "authors.lastName", String.class ) ) ③
                .asArray() ④
                .collector( ProjectionCollector.list() ) ) ⑤
        .where( f -> f.matchAll() )
        .fetchHits( 20 ); ⑥
```

① Call `.object( "authors" )`.

② Define the first inner projection as a projection on the `firstName` field of `authors`.

③ Define the second inner projection as a projection on the `lastName` field of `authors`.

④ Define the result of the projection as an array, meaning the hits will be `Object[]` instances with at index `0` the value of the `firstName` field of `authors` , and at index `1` the value of the `lastName` field of `authors`.

⑤ Define the projection as multivalued, meaning it will yield values of type `List<Object[]>`: one `Object[]` per object in the `authors` object field.

⑥ Each hit will be an instance of `List<Object[]>`: a list containing one array per author, which in turns contains the result of the inner projections, in the given order. Thus, the list of hits will be an instance of `List<List<Object[]>>`.

Projecting to a custom (annotated) type

For more complex object projections, it is possible to define a custom (annotated) record or class and have Hibernate Search infer the corresponding inner projections from the custom type's constructor parameters. This is similar to the projection to a custom (annotated) type through `.select(…)`.

There are a few constraints to keep in mind when annotating a custom projection type:

- The custom projection type must be in the same JAR as entity types, or Hibernate Search will require additional configuration.
- When projecting on value fields or object fields, the path to the projected field is inferred from the constructor parameter name by default, but inference will fail if constructor parameter names are not included in the Java bytecode. Alternatively the path can be provided explicitly through `@FieldProjection(path = …)` /`@ObjectProjection(path = …)`, in which case Hibernate Search won't rely on constructor parameter names.
- When projecting on value fields, the constraints of the `field` projection still apply. In particular, with the Lucene backend, value fields involved in the projection must be configured as projectable.
- When projecting on object fields, the constraints of the `object` projection still apply. In particular, with the Lucene backend, multi-valued object fields involved in the projection must be configured as nested.

Example 15.181: Using a custom record type to project data created from an object field

```
@ProjectionConstructor ①
public record MyAuthorProjection(String firstName, String lastName) { ②
}
```

① Annotate the record type with `@ProjectionConstructor`, either at the type level (if there's only one constructor) or at the constructor level (if there are multiple constructors).

② To project on a value field, add a constructor parameter named after that field and with the same type as that field. See Implicit inner projection inference for more information on how constructor parameters should be defined.
Alternatively, the field projection can be configured explicitly with `@FieldProjection`.

Most projections have a corresponding annotation that can be used on constructor parameters.

```
List<List<MyAuthorProjection>> hits = searchSession.search( Book.class )
        .select( f -> f.object( "authors" ) ①
                .as( MyAuthorProjection.class ) ②
                .collector( ProjectionCollector.list() ) ) ③
        .where( f -> f.matchAll() )
        .fetchHits( 20 ); ④
```

① Call `.object( "authors" )`.

② Define the result of the projection as a custom (annotated) type. Hibernate Search will infer the inner projections from the custom

type's constructor parameters.

③ Each hit will be an instance of the custom projection type, populated with data retrieved from the index.

Custom, non-record classes can also be annotated with `@ProjectionConstructor`, which can be useful if you cannot use records for some reason (for example because you're still using Java 13 or below).

For more information about mapping custom projection types, see Mapping index content to custom types (projection constructors).

`@ObjectProjection` in projections to custom types

To achieve an `object` projection inside a projection to an annotated custom type, you can rely on the default inferred projection: when no annotations are present on a constructor parameter, it will be inferred to an object projection to the field with the same name as the constructor parameter (or a field projection, see here for details).

To force an object projection, or to customize the object projection further (for example to set the field path explicitly), use the `@ObjectProjection` annotation on the constructor parameter:

Example 15.182: Returning custom objects created from an object field within a projection constructor

```
@ProjectionConstructor ①
public record MyBookTitleAndAuthorsProjection(
        @ObjectProjection ②
        List<MyAuthorProjection> authors, ③
        @ObjectProjection(path = "mainAuthor") ④
        MyAuthorProjection theMainAuthor, ⑤
        String title ⑥
) {
}
```

① Annotate the record type with `@ProjectionConstructor`.

② Annotate the parameter that should receive the objects with `@ObjectProjection`.

③ For multi-valued projections, the type of the constructor parameter must be assignable from `List<T>`, where `T` is a type that has a projection constructor itself. See Inner projection and type for more information.

④ Annotation attributes allow customization.

Here we're using a path that is different from the name of the Java constructor parameter.

⑤ For single-valued projections, the type of the constructor parameter must have a projection constructor itself. See Inner projection and type for more information.

⑥ You can of course also declare other parameters with different projections; in this example an inferred projection to the `title` field.

```
List<MyBookTitleAndAuthorsProjection> hits = searchSession.search( Book.class )
        .select( MyBookTitleAndAuthorsProjection.class )①
        .where( f -> f.matchAll() )
        .fetchHits( 20 ); ②
```

① Pass the custom projection type to `.select(…)`.

② Each hit will be an instance of the custom projection type, populated with the requested objects.

The annotation exposes the following attributes:

`path`

The path to the projected field.

If not set, it is inferred from the constructor parameter name.

Field path inference will fail if constructor parameter names are not included in the Java bytecode.

`includePaths`

See `@ObjectProjection` filters to exclude nested projections and break `@ObjectProjection` cycles.

`excludePaths`

See `@ObjectProjection` filters to exclude nested projections and break `@ObjectProjection` cycles.

`includeDepth`

See `@ObjectProjection` filters to exclude nested projections and break `@ObjectProjection` cycles.

For programmatic mapping, use `ObjectProjectionBinder.create()`.

Example 15.183: Programmatic mapping of an `object` projection within a projection constructor

```
TypeMappingStep myBookTitleAndAuthorsProjection =
        mapping.type( MyBookTitleAndAuthorsProjection.class );
myBookTitleAndAuthorsProjection.mainConstructor()
        .projectionConstructor();
myBookTitleAndAuthorsProjection.mainConstructor().parameter( 0 )
        .projection( ObjectProjectionBinder.create() );
myBookTitleAndAuthorsProjection.mainConstructor().parameter( 1 )
        .projection( ObjectProjectionBinder.create( "mainAuthor" ) );
```

`@ObjectProjection` filters to exclude nested projections and break `@ObjectProjection` cycles

By default, `@ObjectProjection` and inferred object projections will include every projection encountered in the projection constructor of the projected type, recursively.

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

- If the projection constructor of the projected type declares many nested projections, only some of which are actually useful to the "surrounding" type, the extra projections will decrease search performance needlessly.
- If there is a cycle of `@ObjectProjection` (e.g. `A` includes a nested object projection `b` of type `B`, which includes a nested projection `a` of type `A`) the root projection type will end up with an infinite amount of fields (`a.b.someField`, `a.b.a.b.someField`, `a.b.a.b.a.b.someField`, …), which Hibernate Search will detect and reject with an exception.

To address these problems, it is possible to apply filters to only include those nested projections that are actually useful. Projections on excluded fields, at runtime, will have their value set to `null`, or an empty list for multivalued projections.

Available filtering attributes on `@ObjectProjection` are:

`includePaths`

The paths of nested index field to be included, i.e. for which the corresponding nested projections will actually be retrieved from the index.

Provided paths must be relative to the projected object field, i.e. they must not include its path.

This takes precedence over `includeDepth` (see below).

Cannot be used in combination with `excludePaths` in the same `@ObjectProjection`.

`excludePaths`

The paths of index fields from the indexed-embedded element that must **not** be embedded.

Provided paths must be relative to the projected object field, i.e. they must not include its path.

This takes precedence over `includeDepth` (see below).

Cannot be used in combination with `includePaths` in the same `@ObjectProjection`.

`includeDepth`

The number of levels of indexed-embedded that will have all their fields included by default.

`includeDepth` is the number of levels of object projections that will have all their nested field/object projections included by default and actually be retrieved from the index.

Up to and including that depth, object projections will be included along with their nested (non-object) field projections, even if these fields are not included explicitly through `includePaths`, unless these fields are excluded explicitly through `excludePaths`:

- `includeDepth=0` means fields of this object projection are **not** included, nor is any field of nested indexed-embedded elements, unless these fields are included explicitly through `includePaths`.

- `includeDepth=1` means fields of this object projection **are** included, unless these fields are excluded explicitly through `excludePaths`, but **not** fields of nested object projections (`@ObjectProjection` within this `@ObjectProjection`), unless these fields are included explicitly through `includePaths`.
- `includeDepth=2` means fields of this object projection and fields of immediately nested object projections (`@ObjectProjection` within this `@ObjectProjection`) **are** included, unless these fields are explicitly excluded through `excludePaths`, but **not** fields of nested object projections beyond that (`@ObjectProjection` within an `@ObjectProjection` within this `@ObjectProjection`), unless these fields are included explicitly through `includePaths`.
- And so on.

The default value depends on the value of the `includePaths` attribute:

- if `includePaths` is empty, `includeDepth` defaults to infinity (include all fields at every level).
- if `includePaths` is **not** empty, `includeDepth` defaults to `0` (only include fields included explicitly).

> *Mixing `includePaths` and `excludePaths` at different nesting levels*
>
> In general, it is possible to use `includePaths` and `excludePaths` at different levels of nested `@ObjectProjection`. When doing so, keep in mind that the filter at each level can only reference reachable paths, i.e. a filter cannot reference a path that was excluded by a nested `@ObjectProjection` (implicitly or explicitly).

Below are three examples: one leveraging `includePaths` only, one leveraging `excludePaths`, and one leveraging `includePaths` and `includeDepth`.

All three examples are based on the following mapped entity that rely on @IndexedEmbedded and the very similar filters it provides:

```
@Entity
@Indexed
public class Human {

    @Id
    private Integer id;

    @FullTextField(analyzer = "name", projectable = Projectable.YES)
    private String name;

    @FullTextField(analyzer = "name", projectable = Projectable.YES)
    private String nickname;

    @ManyToMany
    @IndexedEmbedded(includeDepth = 5, structure = ObjectStructure.NESTED)
    private List<Human> parents = new ArrayList<>();

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

    public Human() {
    }

    // Getters and setters
    // ...

}
```

Example 15.184: Filtering nested projections with `includePaths`

This projection will include the following fields:

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

The following fields in particular are excluded:

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

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

```
@ProjectionConstructor
public record HumanProjection(
        @FieldProjection
        String name,
        @FieldProjection
        String nickname,
        @ObjectProjection(includePaths = { "name", "nickname", "parents.name" })
        List<HumanProjection> parents
) {
}
```

Example 15.185: Filtering nested projections with `excludePaths`

This projection will include the same fields as in the `includePaths` example, but through using the `excludePaths` instead.

This projection will include the following fields:

- `name`
- `nickname`
- `parents.name`: implicitly included because `includeDepth` on `parents` defaults to infinity.
- `parents.nickname`: implicitly included because `includeDepth` on `parents` defaults to infinity.
- `parents.parents.name`: implicitly included because `includeDepth` on `parents` defaults to infinity.

The following fields in particular are excluded:

- `parents.parents.nickname`: **not** included because `excludePaths` explicitly excludes `parents.nickname`.
- `parents.parents.parents`/`parents.parents.parents.<any-field>`: **not** included because `excludePaths` explicitly excludes `parents.parents` stopping any further traversing.

```
@ProjectionConstructor
public record HumanProjection(
        @FieldProjection
        String name,
        @FieldProjection
        String nickname,
        @ObjectProjection(excludePaths = { "parents.nickname", "parents.parents" })
        List<HumanProjection> parents
) {
}
```

Example 15.186: Filtering nested projections with `includePaths` and `includeDepth`

This projection will include the following fields:

- `name`
- `surname`
- `parents.name`: implicitly at depth `0` because `includeDepth > 0` (so `parents.*` is included implicitly).
- `parents.nickname`: implicitly included at depth `0` because `includeDepth > 0` (so `parents.*` is included implicitly).
- `parents.parents.name`: implicitly included at depth `1` because `includeDepth > 1` (so `parents.parents.*` is included implicitly).
- `parents.parents.nickname`: implicitly included at depth `1` because `includeDepth > 1` (so `parents.parents.*` is included implicitly).
- `parents.parents.parents.name`: **not** implicitly included at depth `2` because `includeDepth = 2` (so `parents.parents.parents` is included implicitly, but subfields can only be included explicitly) but explicitly included because `includePaths` on `parents` includes `parents.parents.name`.

The following fields in particular are excluded:

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

- `parents.parents.parents.parents.name`: **not** implicitly included at depth `3` because `includeDepth = 2` (so `parents.parents.parents` is included implicitly, but `parents.parents.parents.parents` and subfields can only be included explicitly) and **not** explicitly included either because `includePaths` on `parents` does not include `parents.parents.parents.name`.

```
@ProjectionConstructor
public record HumanProjection(
        @FieldProjection
        String name,
        @FieldProjection
        String nickname,
        @ObjectProjection(includeDepth = 2, includePaths = { "parents.parents.name" })
        List<HumanProjection> parents
) {
}
```

15.4.12. `constant`: return a provided constant

The `constant` projection returns the same value for every single document, the value being provided when defining the projection.

This is only useful in some edge cases where one wants to include some broader context in the representation of every single hit. In this case, the `constant` value will most likely be used together with a `composite` projection or an `object` projection.

Syntax

Example 15.187: Returning a constant value for every single matched document

```
Instant searchRequestTimestamp = Instant.now();
List<MyPair<Integer, Instant>> hits = searchSession.search( Book.class )
        .select( f -> f.composite()
                .from( f.id( Integer.class ), f.constant( searchRequestTimestamp ) )
                .as( MyPair::new ) )
        .where( f -> f.matchAll() )
        .fetchHits( 20 );
```

In projections to custom types

There is no built-in annotation to use the `constant` projection inside a projection to an annotated custom type.

You can create your own annotation if you need one, backed by a custom projection binder.

15.4.13. `highlight`: return highlighted field values from matched documents

The `highlight` projection returns fragments from full-text fields of matched documents that caused a query match.

In order to use highlight projections of a given field, you need to provide the list of highlighters supported by the field in the mapping.

The highlightable default may already enable the support of highlighting in some cases. For more details see how the `DEFAULT` highlightable value behaves.

Syntax

The `highlight` projection, by default, returns a list of string values per highlighted field, no matter if the field is a single or a multivalued one, since there can be multiple highlighted terms in a field value and depending on the highlighter configuration that can result in multiple text fragments with highlighted terms in them:

Example 15.188: Returning highlights for matched documents

```
List<List<String>> hits = searchSession.search( Book.class )
        .select( f -> f.highlight( "title" ) )
        .where( f -> f.match().field( "title" ).matching( "detective" ) )
        .fetchHits( 20 );
```

As an example, the result may look like:

```
[
    ["The Automatic <em>Detective</em>"], ①
    ["Dirk Gently's Holistic <em>Detective</em> Agency"], ②
    [
      "The Paris <em>Detective</em>",
      "<em>Detective</em> Luc Moncrief series"
    ], ③
]
```

① First hit.
② Second hit.
③ Third hit with multiple highlighted snippets.

In some scenarios, when we know that there is going to be only one highlighted fragment returned, it may be helpful to force the highlight projection to produce a single `String` rather than a `List<String>`. This is only possible when the number of fragments is explicitly set to `1`.

Example 15.189: Forcing a single-valued highlight projection

```
List<String> hits = searchSession.search( Book.class )
        .select( f -> f.highlight( "title" ).single() ) ①
        .where( f -> f.match().field( "title" ).matching( "detective" ) )
        .highlighter( f -> f.unified()
                .numberOfFragments( 1 ) ) ②
        .fetchHits( 20 );
```

① Force the single-valued highlight projection by calling `.single()`.
② Force the number of fragments to `1` so that the highlighter returns a single highlighted fragment at most.

Multivalued fields

Each value of a multivalued field is being highlighted. See how highlighter can be configured to adjust the returned results' behaviour and structure.

At the moment, highlighting a field within a nested object is not supported and attempting to do so will lead to an exception. Highlighting a field within a flattened object will work correctly.

Placing a highlight projection inside an object projection is not supported.

Example 15.190: Returning highlights for flattened objects from matched documents

```
List<List<String>> hits = searchSession.search( Book.class )
        .select( f -> f.highlight( "flattenedAuthors.lastName" ) )
        .where( f -> f.match().field( "flattenedAuthors.lastName" ).matching( "martinez" ) )
        .fetchHits( 20 );
```

Highlighting options

A highlighter can be fine-tuned through its options to change the highlighted output.

Example 15.191: Configuring the default highlighter

```
List<List<String>> hits = searchSession.search( Book.class )
        .select( f -> f.highlight( "title" ) ) ①
        .where( f -> f.match().field( "title" ).matching( "detective" ) )
        .highlighter( f -> f.unified().tag( "<b>", "</b>" ) ) ②
        .fetchHits( 20 );
```

① Building the highlight query as usual.
② Configure the default highlighter. Change the highlighting tag option.

Additionally, if multiple fields are highlighted, and they need different highlighter options then a named highlighter can be used to override the default one.

Example 15.192: Configuring default and named highlighters

```
List<List<?>> hits = searchSession.search( Book.class )
        .select( f -> f.composite().from(
                f.highlight( "title" ),
                f.highlight( "description" ).highlighter( "description-highlighter" ) ①
        ).asList() )
        .where( f -> f.match().field( "title" ).matching( "detective" ) )
        .highlighter( f -> f.unified().tag( "<b>", "</b>" ) ) ②
        .highlighter(
                "description-highlighter",
                f -> f.unified().tag( "<span>", "</span>" )
        ) ③
        .fetchHits( 20 );
```

① Specifying the name of a named highlighter to be applied to the `description` field highlight projection.
② Configure the default highlighter.
③ Configure the named highlighter.

See Highlight DSL for more information on highlighter configuration.

`@HighlightProjection` in projections to custom types

To achieve a `highlight` projection inside a projection to an annotated custom type, use the `@HighlightProjection` annotation on the constructor parameter:

Example 15.193: Returning highlights for matched documents within a projection constructor. Multiple highlighted fragments

```
@ProjectionConstructor ①
public record MyBookTitleAndHighlightedDescriptionProjection(
        @HighlightProjection ②
        List<String> description, ③
        String title ④
) {
}
```

① Annotate the record type with `@ProjectionConstructor`.
② Annotate the parameter that should receive the highlights value with `@HighlightProjection`.
③ The type of the constructor parameter, in this case, must be assignable from `List<String>`, since we are expecting to return multiple highlighted fragments.
④ You can of course also declare other parameters with different projections; in this example an inferred projection to the `title` field.

```
List<MyBookTitleAndHighlightedDescriptionProjection> hits = searchSession.search( Book.class )
        .select( MyBookTitleAndHighlightedDescriptionProjection.class )①
        .where( f -> f.match().field( "description" ).matching( "self-aware" ) )
        .fetchHits( 20 ); ②
```

① Pass the custom projection type to `.select(…)`.
② Each hit will be an instance of the custom projection type, populated with the requested highlights and field.

Example 15.194: Returning highlights for matched documents within a projection constructor. Single highlighted fragment

```
@ProjectionConstructor ①
public record MyBookHighlightedTitleProjection(
        @HighlightProjection ②
        String title, ③
        String description
) {
}
```

① Annotate the record type with `@ProjectionConstructor`.
② Annotate the parameter that should receive the highlights value with `@HighlightProjection`.
③ The type of the constructor parameter, in this case, must be assignable from `String`, since we are planning to return a single highlighted fragment. Note, that only a highlighter that calls the `.numberOfFragments(1)` can be used for single-valued highlight projections.

```
List<MyBookHighlightedTitleProjection> hits = searchSession.search( Book.class )
        .select( MyBookHighlightedTitleProjection.class )①
        .where( f -> f.match().field( "title" ).matching( "robot" ) )
        .highlighter( f -> f.unified().numberOfFragments( 1 ) ) ②
        .fetchHits( 20 ); ③
```

① Pass the custom projection type to `.select(…)`.
② Configure the highlighter to return a single highlighted fragment.
③ Each hit will be an instance of the custom projection type, populated with the requested highlight and field.

The annotation exposes the following attributes:

`path`

The path to the highlighted field.

If not set, it is inferred from the constructor parameter name.

Field path inference will fail if constructor parameter names are not included in the Java bytecode.

`highlighter`

The name of a highlighter configured in the query; useful to apply different options to each highlight projection.

If not set, the projection will use the default highlighter as configured in the query.

For programmatic mapping, use `HighlightProjectionBinder.create()`.

Example 15.195: Programmatic mapping of a `highlight` projection within a projection constructor

```
TypeMappingStep myBookIdAndHighlightedTitleProjection =
        mapping.type( MyBookTitleAndHighlightedDescriptionProjection.class );
myBookIdAndHighlightedTitleProjection.mainConstructor()
        .projectionConstructor();
myBookIdAndHighlightedTitleProjection.mainConstructor().parameter( 0 )
        .projection( HighlightProjectionBinder.create() );
```

Highlight limitations

For now, Hibernate Search has limitations on where highlight projections can be included, and trying to apply highlight projections in these scenarios will lead to an exception being thrown, in particular:

- Such projection cannot be a part of an object projection.

  Example 360. Illegal use of `.highlight(..)` projection within an `.object(..)` projection

  ```
  List<List<?>> hits = searchSession.search( Book.class )
          .select( f -> f.object( "authors" )
                  .from(
                          f.highlight( "authors.firstName" ),
                          f.highlight( "authors.lastName" )
                  ).asList()
          )
          .where( f -> f.match().field( "authors.firstName" ).matching( "Art*" ) )
          .fetchHits( 20 );
  ```

  Doing so **will** lead to an **exception**.

- Fields of an object with a nested structure cannot be highlighted under any circumstances.

  Example 361. Illegal use of `.highlight(..)` projection within an `.object(..)` projection

  ```
  List<?> hits = searchSession.search( Book.class )
          .select( f -> f.highlight( "authors.firstName" ) )
          .where( f -> f.match().field( "authors.firstName" ).matching( "Art*" ) )
          .fetchHits( 20 );
  ```

  Assuming that `authors` are mapped as nested structure, e.g.:

  ```
  @IndexedEmbedded(structure = ObjectStructure.NESTED)
  private List<Author> authors = new ArrayList<>();
  ```

  Trying to apply such projection **will** result in an **exception** being thrown.

These limitations should be addressed by HSEARCH-4841.

15.4.14. `withParameters`: create projections using query parameters

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

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

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

The `withParameters` projection allows building projections using query parameters.

This type of projection requires a function that accepts query parameters and returns a projection. That function will get called at query building time.

Syntax

The `withParameters` projection return type depends on the projection type configured within the `.withParameters(..)`:

Example 15.196: Creating a projection with query parameters

```
GeoPoint center = GeoPoint.of( 47.506060, 2.473916 );
SearchResult<Double> result = searchSession.search( Author.class )
        .select( f -> f.withParameters( params -> f ①
                .distance( "placeOfBirth", params.get( "center", GeoPoint.class ) ) ) ) ②
        .where( f -> f.matchAll() )
        .param( "center", center ) ③
        .fetch( 20 );
```

① Start creating the `.withParameters()` projection.
② Access the query parameter `center` of `GeoPoint` type when constructing the projection.
③ Set parameters required by the projection at the query level.

15.4.15. Backend-specific extensions

By calling `.extension(…)` while building a query, it is possible to access backend-specific projections.

As their name suggests, backend-specific projections are not portable from one backend technology to the other.

Lucene: `document`

The `.document()` projection returns the matched document as a native Lucene `Document`.

This feature implies that application code rely on Lucene APIs directly.

An upgrade of Hibernate Search, even for a bugfix (micro) release, may require an upgrade of Lucene, which may lead to breaking API changes in Lucene.

If this happens, you will need to change application code to deal with the changes.

Syntax

Example 15.197: Returning the matched document as a native `org.apache.lucene.document.Document`

```
List<Document> hits = searchSession.search( Book.class )
        .extension( LuceneExtension.get() )
        .select( f -> f.document() )
        .where( f -> f.matchAll() )
        .fetchHits( 20 );
```

The returned document is similar to the one that was indexed.

For example, if a value was transformed by the `toIndexedValue` method of a value bridge upon indexing, this transformed value (after encoding) will be the value included in the document: Hibernate Search will not convert it back using `ValueBridge#fromIndexedValue`.

However, there are some differences between the returned document and the one that was indexed:

- Only stored fields are present.
- Even stored fields may not have the same `FieldType` as they originally had.
- The document structure flattened, i.e. even fields from nested documents are all added to same returned document.
- Dynamic fields may be missing.

If you want a projection that retrieves the value as it was in your entity upon indexing, use a `field` projection.

In projections to custom types

There is no built-in annotation to use the `document` projection inside a projection to an annotated custom type.

You can create your own annotation if you need one, backed by a custom projection binder.

Lucene: `documentTree`

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

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

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

The `.documentTree()` projection returns the matched document as a tree containing native Lucene `Document` and corresponding nested tree nodes.

This feature implies that application code rely on Lucene APIs directly.

An upgrade of Hibernate Search, even for a bugfix (micro) release, may require an upgrade of Lucene, which may lead to breaking API changes in Lucene.

If this happens, you will need to change application code to deal with the changes.

Syntax

Example 15.198: Returning the matched document as a `DocumentTree`

```
List<DocumentTree> hits = searchSession.search( Book.class )
        .extension( LuceneExtension.get() )
        .select( f -> f.documentTree() )
        .where( f -> f.matchAll() )
        .fetchHits( 20 );
```

The documents returned in this tree have the same characteristics as the documents returned by the `.document()` projection, with the exception of not being flattened.

In projections to custom types

There is no built-in annotation to use the `documentTree` projection inside a projection to an annotated custom type.

You can create your own annotation if you need one, backed by a custom projection binder.

Lucene: `explanation`

The `.explanation()` projection returns an explanation of the match as a native Lucene `Explanation`.

Regardless of the API used, explanations are rather costly performance-wise: only use them for debugging purposes.

This feature implies that application code rely on Lucene APIs directly.

An upgrade of Hibernate Search, even for a bugfix (micro) release, may require an upgrade of Lucene, which may lead to breaking API changes in Lucene.

If this happens, you will need to change application code to deal with the changes.

Syntax

Example 15.199: Returning the score explanation as a native `org.apache.lucene.search.Explanation`

```
List<Explanation> hits = searchSession.search( Book.class )
        .extension( LuceneExtension.get() )
        .select( f -> f.explanation() )
        .where( f -> f.matchAll() )
        .fetchHits( 20 );
```

In projections to custom types

There is no built-in annotation to use the `explanation` projection inside a projection to an annotated custom type.

You can create your own annotation if you need one, backed by a custom projection binder.

Elasticsearch: `source`

The `.source()` projection returns the JSON of the document as it was indexed in Elasticsearch, as a `JsonObject`.

This feature requires to directly manipulate JSON in application code.

The syntax of this JSON may change:

- when you upgrade the underlying Elasticsearch cluster to the next version;
- when you upgrade Hibernate Search to the next version, even for a bugfix (micro) release.

If this happens, you will need to change application code to deal with the changes.

Syntax

Example 15.200: Returning the matched document source as a `JsonObject`

```
List<JsonObject> hits = searchSession.search( Book.class )
        .extension( ElasticsearchExtension.get() )
        .select( f -> f.source() )
        .where( f -> f.matchAll() )
        .fetchHits( 20 );
```

The source is returned exactly as it appears in the response from Elasticsearch. In particular, Hibernate Search will not apply any kind of the conversions described in Type of projected values.

For example, if a value was transformed by the `toIndexedValue` method of a value bridge upon indexing, this transformed value will be the value included in the source: Hibernate Search will not convert it back using `ValueBridge#fromIndexedValue`.

If you want a projection that retrieves the value as it was in your entity upon indexing, use a `field` projection.

In projections to custom types

There is no built-in annotation to use the `source` projection inside a projection to an annotated custom type.

You can create your own annotation if you need one, backed by a custom projection binder.

Elasticsearch: `explanation`

The `.explanation()` projection returns an explanation of the match as a `JsonObject`.

Regardless of the API used, explanations are rather costly performance-wise: only use them for debugging purposes.

This feature requires to directly manipulate JSON in application code.

The syntax of this JSON may change:

- when you upgrade the underlying Elasticsearch cluster to the next version;
- when you upgrade Hibernate Search to the next version, even for a bugfix (micro) release.

If this happens, you will need to change application code to deal with the changes.

Syntax

Example 15.201: Returning the score explanation as a `JsonObject`

```
List<JsonObject> hits = searchSession.search( Book.class )
        .extension( ElasticsearchExtension.get() )
        .select( f -> f.explanation() )
        .where( f -> f.matchAll() )
        .fetchHits( 20 );
```

In projections to custom types

There is no built-in annotation to use the `explanation` projection inside a projection to an annotated custom type.

You can create your own annotation if you need one, backed by a custom projection binder.

Elasticsearch: `jsonHit`

The `.jsonHit()` projection returns the exact JSON returned by Elasticsearch for the hit, as a `JsonObject`.

This is particularly useful when customizing the request's JSON to ask for additional data within each hit.

This feature requires to directly manipulate JSON in application code.

The syntax of this JSON may change:

- when you upgrade the underlying Elasticsearch cluster to the next version;
- when you upgrade Hibernate Search to the next version, even for a bugfix (micro) release.

If this happens, you will need to change application code to deal with the changes.

Syntax

Example 15.202: Returning the Elasticsearch hit as a `JsonObject`

```
List<JsonObject> hits = searchSession.search( Book.class )
        .extension( ElasticsearchExtension.get() )
        .select( f -> f.jsonHit() )
        .where( f -> f.matchAll() )
        .fetchHits( 20 );
```

In projections to custom types

There is no built-in annotation to use the `jsonHit` projection inside a projection to an annotated custom type.

You can create your own annotation if you need one, backed by a custom projection binder.

15.5. Highlight DSL

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

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

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

15.5.1. Basics

Highlighting is a projection that returns fragments from full-text fields of matched documents that caused a query match. Specific terms that caused the match are "highlighted" with a pair of opening and closing tags. It can help a user to quickly identify the information they were searching for on a results page.

Highlight projections are only available for full-text fields with an attribute configuration that allows it:

Example 15.203: Configuring fields for highlighting

```
@Entity(name = Book.NAME)
@Indexed
public class Book {

    public static final String NAME = "Book";

    @Id
    private Integer id;

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

    @FullTextField(analyzer = "english",
            highlightable = { Highlightable.PLAIN, Highlightable.UNIFIED }) ②
    private String title;
```

```
    @FullTextField(analyzer = "english",
            highlightable = Highlightable.ANY) ③
    @Column(length = 10000)
    private String description;

    @FullTextField(analyzer = "english",
            projectable = Projectable.YES,
            termVector = TermVector.WITH_POSITIONS_OFFSETS) ④
    @Column(length = 10000)
    @ElementCollection
    private List<String> text;

    @GenericField ⑤
    @Column(length = 10000)
    @ElementCollection
    private List<String> keywords;


}
```

① A regular full-text field. Such field can allow highlight projections if the Elasticsearch backend is used. See the definition of the default highlightable for any details.

② A full-text field that explicitly allows for plain and unified highlighters to be applied to it.

③ A full-text field that explicitly allows any highlighter types to be applied to it. See the definition of the `ANY` highlightable for further details.

④ A full-text field that implicitly allows any highlighter types to be applied to it. Allowing projection and setting the term vector storage strategy to `WITH_POSITIONS_OFFSETS` implies that any highlighter type can be used to create a highlight projection with both Lucene or Elasticsearch backends.

⑤ A generic text field—such field will not allow highlight projections.

Example 15.204: Using a highlight projection

```
SearchSession searchSession = /* ... */ ①

List<List<String>> result = searchSession.search( Book.class ) ②
        .select( f -> f.highlight( "title" ) ) ③
        .where( f -> f.match().field( "title" ).matching( "mystery" ) ) ④
        .fetchHits( 20 ); ⑤
```

① Retrieve the `SearchSession`.

② Start building the query as usual.

③ Mention that the expected result of the query is the highlights on the "title" field. An exception will be thrown if the field is not a full-text field with highlighting enabled or if it does not exist.

④ The predicate from the where clause will be used to determine which documents to include in the results and highlight the text from the "title" field.

Note that to get a nonempty list for a highlight projection, the field we apply such projection to should be a part of a predicate. If there's no match within the field we want to highlight, be it because the document was added to the results because of some other predicate condition, or because the highlighted field wasn't a part of a predicate at all, an empty list will be returned by default. See no match size configuration option for more details on how this can be adjusted.

⑤ Fetch the results, which will have highlighted fragments.

Note that the result of applying a highlight projection is always a list of `String`.

As an example, the result may look like:

```
[
    ["The Boscombe Valley <em>Mystery</em>"], ①
    [
      "A Caribbean <em>Mystery</em>",
      "Miss Marple: A Caribbean <em>Mystery</em> by Agatha Christie"
    ], ②
    ["A <em>Mystery</em> of <em>Mysteries</em>: The Death and Life of Edgar Allan Poe"] ③
]
```

① First hit.

② Second hit with multiple highlighted snippets.
③ Third hit.

Highlight projections, just like field projections, can also be used in a combination with other projection types as well as with other highlight projections:

Example 15.205: Using a composite highlight projection

```
List<List<?>> result = searchSession.search( Book.class ) ①
        .select( f -> f.composite().from(
                f.id(), ②
                f.field( "title", String.class ), ③
                f.highlight( "description" ) ④
        ).asList() )
        .where( f -> f.match().fields( "title", "description" ).matching( "scandal" ) ) ⑤
        .fetchHits( 20 ); ⑥
```

① Start building the query as usual.
② Add an id projection.
③ Add a regular field projection on a "title" field.
④ Add a highlight projection on a "description" field.
⑤ Provide a predicate to filter documents and to use to highlight the results.
⑥ Fetch the results.

A highlighter behavior can be configured. See various available configuration options. A highlighter definition is provided after a where clause of a query:

Example 15.206: Configuring a default highlighter

```
List<List<?>> result = searchSession.search( Book.class )
        .select( f -> f.composite().from(
                f.highlight( "title" ),
                f.highlight( "description" )
        ).asList() )
        .where( f -> f.match().fields( "title", "description" ).matching( "scandal" ) ) ①
        .highlighter( f -> f.plain().noMatchSize( 100 ) ) ②
        .fetchHits( 20 ); ③
```

① Specify a predicate that would look for matches in both `title` and `description` fields.
② Specify the details of the default highlighter. Setting the no match size to a positive value to let the highlighter know that we want to get some text back even if there will be nothing to highlight in a particular field.
③ Fetch the results.

15.5.2. Highlighter type

Before a highlighter can be configured, you need to pick its type. Picking the highlighter type is the first step in a highlighter definition:

Example 15.207: Specifying the plain highlighter type

```
searchSession.search( Book.class )
        .select( f -> f.highlight( "title" ) )
        .where( f -> f.match().fields( "title", "description" ).matching( "scandal" ) )
        .highlighter( f -> f.plain() /* ... */ ) ①
        .fetchHits( 20 );
```

① Starting the definition of a plain highlighter.

Example 15.208: Specifying the unified highlighter type

```
searchSession.search( Book.class )
```

```
        .select( f -> f.highlight( "title" ) )
        .where( f -> f.match().fields( "title", "description" ).matching( "scandal" ) )
        .highlighter( f -> f.unified() /* ... */ ) ①
        .fetchHits( 20 );
```

① Starting the definition of a unified highlighter.

Example 15.209: Specifying the fast vector highlighter type

```
searchSession.search( Book.class )
        .select( f -> f.highlight( "description" ) )
        .where( f -> f.match().fields( "title", "description" ).matching( "scandal" ) )
        .highlighter( f -> f.fastVector() /* ... */ ) ①
        .fetchHits( 20 );
```

① Starting the definition of a fast vector highlighter.

There are three options to choose from when it comes to the highlighter type:

Plain

The plain highlighter can be useful for simple queries targeting a single field on a small number of documents. This highlighter uses a standard Lucene highlighter. It reads the string value of a highlighted field, then creates a small in-memory index from it and applies query logic to perform the highlighting.

Unified

The unified highlighter is used by default and does not necessarily rely on re-analyzing the text, as it can get the offsets either from postings or from term vectors.

This highlighter uses a break iterator (breaks the text into sentences by default) to break the text into later scored passages. It better supports more complex queries. Since it can work with prebuilt data, it performs better in case of a larger amount of documents compared to the plain highlighter.

Fast vector

The fast vector highlighter, in addition to using a break iterator similar to the unified highlighter, it can use the boundary characters to control the highlighted snippet.

This is the only highlighter that can assign different weights to highlighted fragments, allowing it to show a fragment score differences by wrapping it with a different tag. For more on tags, see the corresponding section.

The fast vector highlighter is also the one which can highlight entire matched phrases. Using phrase predicates with other highlighter types will lead to each word in a phrase being highlighted separately.

15.5.3. Named highlighters

Sometimes we might want to apply different highlighters to various fields. We have already seen that a highlighter can be configured. The highlighter from that example is called the default highlighter. Search queries also allow to configure named highlighters. A named highlighter has the same configuration capabilities as the default one. It overrides the options set by the default highlighter if such was configured. If a default highlighter was configured for a query then every named highlighter configured on the same query must be of the same type as the default one. Mixing various highlighter types within the same query is only allowed when no default highlighter was configured.

When a highlight projection has a named highlighter passed to an optional `highlighter(..)` call chained as a part of the highlight projection definition, then that particular highlighter will be applied to a field projection. Named highlighters can be reused withing a query, i.e. the same name of a named highlighter can be passed to multiple highlight projections.

Example 15.210: Configuring both default and named highlighters

```
List<List<?>> result = searchSession.search( Book.class )
        .select( f -> f.composite().from(
                f.highlight( "title" ), ①
                f.highlight( "description" ).highlighter( "customized-plain-highlighter" ) ②
        ).asList() )
        .where( f -> f.match().fields( "title", "description" ).matching( "scandal" ) )
        .highlighter( f -> f.plain().tag( "<b>", "</b>" ) ) ③
        .highlighter( "customized-plain-highlighter", f -> f.plain().noMatchSize( 100 ) ) ④
```

```
        .fetchHits( 20 ); ⑤
```

① Add the highlight projection on a "title" field. This projection uses the default highlighter.
② Add the highlight projection on a "description" field and specify the name of the named highlighter.
③ Specify the details of the default highlighter.
④ Specify the details of the named highlighter. Note that the name matches the name passed to the configuration of the "description" highlight projection.
⑤ Fetch the results.

The name of a named highlighter cannot be `null` or an empty string. An exception will be thrown if such values are used.

15.5.4. Tags

By default, the highlighted text is wrapped with a pair of `<em>`/`</em>` tags. A custom pair of tags can be provided to change this behaviour. Usually, tags are a pair of HTML tags, but they can be a pair of any character sequences.

Example 15.211: Setting custom tags

```
List<List<String>> result = searchSession.search( Book.class )
        .select( f -> f.highlight( "title" ) )
        .where( f -> f.match().fields( "title" ).matching( "scandal" ) )
        .highlighter( f -> f.unified().tag( "<strong>", "</strong>" ) ) ①
        .fetchHits( 20 );
```

① Passing a pair of open/close tags that will be used to highlight the text.

The fast vector highlighter, which can handle multiple tags, has a few additional methods that accept a collection of tags.

Example 15.212: Setting multiple custom tags

```
result = searchSession.search( Book.class )
        .select( f -> f.highlight( "description" ) )
        .where( f -> f.match().fields( "description" ).matching( "scandal" ) )
        .highlighter( f -> f.fastVector()
                .tags( ①
                        Arrays.asList( "<em class=\"class1\">", "<em class=\"class2\">" ),
                        "</em>"
                ) )
        .fetchHits( 20 );
result = searchSession.search( Book.class )
        .select( f -> f.highlight( "description" ) )
        .where( f -> f.match().fields( "description" ).matching( "scandal" ) )
        .highlighter( f -> f.fastVector()
                .tags( ②
                        Arrays.asList( "<em>", "<strong>" ),
                        Arrays.asList( "</em>", "</strong>" )
                ) )
        .fetchHits( 20 );
```

① Passing a collection of open tags and a single closing tag that will be used to highlight the text. It can be helpful when configuring a tag schema that differs only in an attribute of an opening tag.
② Passing a pair of collections containing open/close tags that will be used to highlight the text.

Additionally, a fast vector highlighter has the option to enable a tag schema and set it to `HighlighterTagSchema.STYLED` to use a predefined set of tags.

Example 15.213: Setting a styled tags schema

```
List<List<String>> result = searchSession.search( Book.class )
        .select( f -> f.highlight( "description" ) )
        .where( f -> f.match().fields( "description" ).matching( "scandal" ) )
        .highlighter( f -> f.fastVector()
```

```
                .tagSchema( HighlighterTagSchema.STYLED ) ①
        )
        .fetchHits( 20 );
```

① Passing a styled tags schema that will be used to highlight the text.

Using a styled tags schema is just a shortcut to defining tags as:

Example 15.214: Setting tags as if the styled tags schema is used

```
List<List<String>> result = searchSession.search( Book.class )
        .select( f -> f.highlight( "description" ) )
        .where( f -> f.match().fields( "description" ).matching( "scandal" ) )
        .highlighter( f -> f.fastVector()
                .tags( Arrays.asList(
                        "<em class=\"hlt1\">",
                        "<em class=\"hlt2\">",
                        "<em class=\"hlt3\">",
                        "<em class=\"hlt4\">",
                        "<em class=\"hlt5\">",
                        "<em class=\"hlt6\">",
                        "<em class=\"hlt7\">",
                        "<em class=\"hlt8\">",
                        "<em class=\"hlt9\">",
                        "<em class=\"hlt10\">"
                ), "</em>" ) ①
        )
        .fetchHits( 20 );
```

① Passing the same collection of tags used when a styled schema is applied.

Calling different tags configuration methods (`tag(..)`/`tags(..)`/`tagSchema(..)` or the same one multiple times within the same highlighter definition will **not** combine them. Tags set by the last call will be applied.

15.5.5. Encoder

Encoding can be applied to the highlighted snippets when highlighting the fields that store HTML. Applying an HTML encoder to a highlighter will encode the text for inclusion into an HTML document: it will replace HTML meta-characters such as < with their entity equivalent such as `<`; however it will not escape the highlighting tags. By default, a `HighlighterEncoder.DEFAULT` encoder is used, which keeps the text as is.

Example 15.215: Setting the HTML encoder

```
List<List<String>> result = searchSession.search( Book.class )
        .select( f -> f.highlight( "title" ) )
        .where( f -> f.match().fields( "title" ).matching( "scandal" ) )
        .highlighter( f -> f.unified().encoder( HighlighterEncoder.HTML ) ) ①
        .fetchHits( 20 );
```

① Configuring the HTML encoder.

15.5.6. No match size

In case of more complex queries or when highlighting is performed for multiple fields, it might lead to a situation where the query matched a document, but a particular highlighted field did not contribute to that match. This will lead to an empty list of highlights for that particular document and that field. No match size option allows you to still get some text returned even if the field didn't contribute to the document match, and there's nothing to be highlighted in it.

The number set by this property defines the number of characters to be included starting at the beginning of a field. Depending on the highlighter type, the amount of text returned might not precisely match the configured value since highlighters usually try not to break the text in the middle of a word/sentence, depending on their configuration. By default, this option is set to `0` and text will only be returned if there's something to highlight.

Example 15.216: Setting the no match size

```
List<List<String>> result = searchSession.search( Book.class )
        .select( f -> f.highlight( "description" ) )
        .where( f -> f.bool()
                .must( f.match().fields( "title" ).matching( "scandal" ) ) ①
                .should( f.match().fields( "description" ).matching( "scandal" ) ) ②
        )
        .highlighter( f -> f.fastVector().noMatchSize( 100 ) ) ③
        .fetchHits( 20 );
```

① We are looking for matches in the title.

② In case a word can also be found in the description, we'd want it to be highlighted.

③ Setting the no match size to `100` to still get at least `100` first characters of a description even if the match for a word we are searching for is not found.

The unified highlighter from the Lucene backend has a limited support for this option. It cannot limit the amount of the returned text, and works more like a boolean flag to enable/disable the feature. If a highlighter of this type has the option not set or set to `0` then no text is returned when there was no match found. Otherwise, if the option for a highlighter of this type was set to a positive integer, all text is returned, no matter the actual value.

15.5.7. Fragment size and number of fragments

The fragment size sets the amount of text included in each highlighted fragment, by default `100` characters.

This is not a "hard" limit, since highlighters usually try not to break the fragment in the middle of a word. Additionally, other features such as boundary scanning may lead to more text before and after the fragment being included as well.

A number of fragments configuration sets the maximum number of strings included in the resulting highlighted list. By default, the number of fragments is limited to `5`.

A combination of these options can be helpful when highlighting large text fields.

Example 15.217: Setting the fragment size and the number of fragments

```
List<List<String>> result = searchSession.search( Book.class )
        .select( f -> f.highlight( "description" ) )
        .where( f -> f.match().fields( "description" ).matching( "king" )
        )
        .highlighter( f -> f.fastVector()
                .fragmentSize( 50 ) ①
                .numberOfFragments( 2 ) ②
        )
        .fetchHits( 20 );
```

① Configuring the fragment size.

② Configuring the maximum number of fragments to be returned.

These options are supported by all highlighter types on the Elasticsearch backend. As for the Lucene backend — the number of fragments is also supported by all highlighter types, while only plain and fast-vector highlighters support fragment size.

15.5.8. Order

By default, highlighted fragments are returned in the order of occurrence in the text. By enabling the order by score option most relevant fragments will be returned at the top of the list.

Example 15.218: Setting the fragment size and number of fragments

```
List<List<String>> result = searchSession.search( Book.class )
        .select( f -> f.highlight( "description" ) )
```

```
        .where( f -> f.bool() ①
                .should( f.match().fields( "description" ).matching( "king" ) )
                .should( f.match().fields( "description" ).matching( "souvenir" ).boost( 10.0f ) )
        )
        .highlighter( f -> f.fastVector().orderByScore( true ) ) ②
        .fetchHits( 20 );
```

① A query that would boost the match of one of the searched words.
② Configuring the order by the score to be enabled.

15.5.9. Fragmenter

By default, the plain highlighter breaks up text into same-sized fragments but tries to avoid breaking up a phrase to be highlighted. This is the behaviour of the `HighlighterFragmenter.SPAN` fragmenter. Alternatively, fragmenter can be set to `HighlighterFragmenter.SIMPLE` that simply breaks up the text into same-sized fragments.

Example 15.219: Setting the fragmenter

```
List<List<String>> result = searchSession.search( Book.class )
        .select( f -> f.highlight( "description" ) )
        .where( f -> f.match().fields( "description" ).matching( "souvenir" ) )
        .highlighter( f -> f.plain().fragmenter( HighlighterFragmenter.SIMPLE ) ) ①
        .fetchHits( 20 );
```

① Configuring the simple fragmenter.

This option is supported only by the plain highlighter.

15.5.10. Boundary scanner

Unified and fast vector highlighters use boundary scanners to create highlighted fragments: they try to expand highlighted fragments by scanning text before and after those fragments for word/sentence boundaries.

An optional locale parameter can be supplied to specify how to search for sentence and word boundaries. A sentence boundary scanner is a default option for the unified highlighter.

There are two ways to supply boundary scanner configuration to a highlighter.

Example 15.220: Setting the boundary scanner with DSL

```
List<List<String>> result = searchSession.search( Book.class )
        .select( f -> f.highlight( "description" ) )
        .where( f -> f.match().fields( "description" ).matching( "king" ) )
        .highlighter( f -> f.fastVector()
                .boundaryScanner() ①
                        .word() ②
                        .locale( Locale.ENGLISH ) ③
                        .end() ④
                /* ... */ ⑤
        )
        .fetchHits( 20 );
```

① Start the definition of a boundary scanner.
② Pick a boundary scanner type—a word scanner in this case.
③ Set an optional locale.
④ End the definition of a boundary scanner.
⑤ Set any other options.

Example 15.221: Setting the boundary scanner using lambda

```
List<List<String>> result = searchSession.search( Book.class )
        .select( f -> f.highlight( "description" ) )
```

```
        .where( f -> f.match().fields( "description" ).matching( "king" ) )
        .highlighter( f -> f.fastVector()
                .boundaryScanner(
                        bs -> bs.word() ①
                )
                /* ... */ ②
        )
        .fetchHits( 20 );
```

① Pass a lambda configurer to set up a boundary scanner.
② Set any other options

Alternatively, a fast vector highlighter can use a character boundary scanner which relies on two other configurations—boundary characters and boundary max scan. When a character boundary scanner is used after a highlighted fragment is formed with highlighted text centred, the highlighter checks for the first occurrence of any configured boundary characters to the left and right of a currently created fragment. This lookup happens only for a maximum number of characters configured by the boundary max scan option. If no boundary characters are found, no additional text will be included besides the already highlighted phrase with surrounding text based on a fragment size option set for a highlighter.

The default list of boundary characters includes `.,!? \t\n`. The default boundary max scan is equal to `20` characters.

Character boundary scanner is a default option for the fast vector highlighters.

Example 15.222: Setting the character boundary scanner

```
List<List<String>> result = searchSession.search( Book.class )
        .select( f -> f.highlight( "description" ) )
        .where( f -> f.match().fields( "description" ).matching( "scene" ) )
        .highlighter( f -> f.fastVector()
                .boundaryScanner() ①
                        .chars() ②
                        .boundaryChars( "\n" ) ③
                        .boundaryMaxScan( 1000 ) ④
                        .end() ⑤
                /* ... */ ⑥
        )
        .fetchHits( 20 );
```

Assuming that we have a text that contains multiple paragraphs separated with a new line (`\n`), we want to get the entire paragraph containing the highlighted phrase. To do so, boundary characters will be set to `\n`, and the max scan option will be based on the number of characters in paragraphs.

① Start the definition of a boundary scanner.
② Pick a boundary scanner type—a character scanner.
③ Set a string containing boundary characters. The overloaded methods can accept a `String` or a `Character` array.
④ Set the max scan option.
⑤ End the definition of a boundary scanner.
⑥ Set any other options.

This option is supported by the unified and fast vector highlighter types.

15.5.11. Phrase limit

Phrase limit allows specifying the maximum number of matching phrases in a document for highlighting. The highlighter will be going through the text and as soon as it reaches the maximum number of highlighted phrases it'll stop leaving any further occurrences as not highlighted.

This limit is different from the maximum number of fragments:

1. Fragments are the strings returned by the highlight projections, while phrases are the sequences of highlighted (matching) terms in each fragment. A fragment may include multiple highlighted phrases, and a given phrase may appear in multiple fragments.
2. The phrase limit is about limiting the highlighting of occurrences of matched phrases, be it multiple occurrences of the same phrase or a mix of different phrases. For example, if we were to search for `fox` and `dog` occurrences in the

> sentence `The quick brown fox jumps over the lazy dog` and set the phrase limit to `1`, then we'll have only `fox` being highlighted since it was the first match in the text and the phrase limit was reached.

By default, this phrase limit is equal to `256`.

This option can be helpful if a field contains many matches and has a lot of text overall, but we are not interested in highlighting every occurrence.

Example 15.223: Setting the phrase limit

```
List<List<String>> result = searchSession.search( Book.class )
        .select( f -> f.highlight( "description" ) )
        .where( f -> f.match().fields( "description" ).matching( "bank" ) )
        .highlighter( f -> f.fastVector()
                .phraseLimit( 1 ) ①
        )
        .fetchHits( 20 );
```

① Configuring the phrase limit.

> This option is supported only by the fast vector highlighter type.

15.6. Aggregation DSL

15.6.1. Basics

Sometimes, you don't just need to list query hits directly: you also need to group and aggregate the hits.

For example, almost any e-commerce website you can visit will have some sort of "faceting", which is a simple form of aggregation. In the "book search" webpage of an online bookshop, beside the list of matching books, you will find "facets", i.e. a count of matching documents in various categories. These categories can be taken directly from the indexed data, e.g. the genre of the book (science-fiction, crime fiction, …), but also derived from the indexed data slightly, e.g. a price range ("less than $5", "less than $10", …).

Aggregations allow just that (and, depending on the backend, much more): they allow the query to return "aggregated" hits.

Aggregations can be configured when building the search query:

Example 15.224: Defining an aggregation in a search query

```
SearchSession searchSession = /* ... */ ①

AggregationKey<Map<Genre, Long>> countsByGenreKey = AggregationKey.of( "countsByGenre" ); ②

SearchResult<Book> result = searchSession.search( Book.class ) ③
        .where( f -> f.match().field( "title" ) ④
                .matching( "robot" ) )
        .aggregation( countsByGenreKey, f -> f.terms() ⑤
                .field( "genre", Genre.class ) )
        .fetch( 20 ); ⑥

Map<Genre, Long> countsByGenre = result.aggregation( countsByGenreKey ); ⑦
```

① Retrieve the `SearchSession`.
② Define a key that will uniquely identify the aggregation. Make sure to give it the correct type (see <6>).
③ Start building the query as usual.
④ Define a predicate: the aggregation will only take into account documents matching this predicate.
⑤ Request an aggregation on the `genre` field, with a separate count for each genre: science-fiction, crime fiction, … If the field does not exist or cannot be aggregated, an exception will be thrown.
⑥ Fetch the results.
⑦ Retrieve the aggregation from the results as a `Map`, with the genre as key and the hit count as value of type `Long`.

Alternatively, if you don't want to use lambdas:

Example 15.225: Defining an aggregation in a search query — object-based syntax

```
SearchSession searchSession = /* ... */

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

AggregationKey<Map<Genre, Long>> countsByGenreKey = AggregationKey.of( "countsByGenre" );

SearchResult<Book> result = searchSession.search( scope )
        .where( scope.predicate().match().field( "title" )
                .matching( "robot" )
                .toPredicate() )
        .aggregation( countsByGenreKey, scope.aggregation().terms()
                .field( "genre", Genre.class )
                .toAggregation() )
        .fetch( 20 );

Map<Genre, Long> countsByGenre = result.aggregation( countsByGenreKey );
```

In order to use aggregations based on the value of a given field, you need to mark the field as aggregable in the mapping.

This is not possible for full-text fields, in particular; see here for an explanation and some solutions.

Faceting generally involves a concept of "drill-down", i.e. the ability to select a facet and restrict the hits to only those that match that facet.

Hibernate Search 5 used to offer a dedicated API to enable this "drill-down", but in Hibernate Search 6 you should simply create a new query with the appropriate predicate.

The aggregation DSL offers more aggregation types, and multiple options for each type of aggregation. To learn more about the `terms` aggregation, and all the other types of aggregations, refer to the following sections.

15.6.2. `terms`: group by the value of a field

The `terms` aggregation returns a count of documents for each term value of a given field.

In order to use aggregations based on the value of a given field, you need to mark the field as aggregable in the mapping.

This is not possible for full-text fields, in particular; see here for an explanation and some solutions.

The `terms` aggregation is not available on geo-point fields.

Example 15.226: Counting hits grouped by the value of a field

```
AggregationKey<Map<Genre, Long>> countsByGenreKey = AggregationKey.of( "countsByGenre" );
SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .aggregation( countsByGenreKey, f -> f.terms()
                .field( "genre", Genre.class ) ) ①
        .fetch( 20 );
Map<Genre, Long> countsByGenre = result.aggregation( countsByGenreKey ); ②
```

① Define the path and type of the field whose values should be considered.
② The result is a map from field value to document count.

Skipping conversion

By default, the values returned by the `terms` aggregation have the same type as the entity property corresponding to the target field.

For example, if an entity property if of an enum type, the corresponding field may be of type `String`; the values returned by the `terms` aggregation will be of the enum type regardless.

This should generally be what you want, but if you ever need to bypass conversion and have unconverted values returned to you instead (of type `String` in the example above), you can do it this way:

Example 15.227: Counting hits grouped by the value of a field, without converting field values

```
AggregationKey<Map<String, Long>> countsByGenreKey = AggregationKey.of( "countsByGenre" );
SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .aggregation( countsByGenreKey, f -> f.terms()
                .field( "genre", String.class, ValueModel.INDEX ) )
        .fetch( 20 );
Map<String, Long> countsByGenre = result.aggregation( countsByGenreKey );
```

See Type of projected values for more information.

`maxTermCount`: limiting the number of returned entries

By default, Hibernate Search will return at most 100 entries. You can customize the limit by calling `.maxTermCount(…)`:

Example 15.228: Setting the maximum number of returned entries in a `terms` aggregation

```
AggregationKey<Map<Genre, Long>> countsByGenreKey = AggregationKey.of( "countsByGenre" );
SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .aggregation( countsByGenreKey, f -> f.terms()
                .field( "genre", Genre.class )
                .maxTermCount( 1 ) )
        .fetch( 20 );
Map<Genre, Long> countsByGenre = result.aggregation( countsByGenreKey );
```

`minDocumentCount`: requiring at least N matching documents per term

By default, Hibernate search will return an entry only if the document count is at least 1.

You can set the threshold to an arbitrary value by calling `.minDocumentCount(…)`.

This is particularly useful to return all terms that exist in the index, even if no document containing the term matched the query. To that end, just call `.minDocumentCount(0)`:

Example 15.229: Including values from unmatched documents in a `terms` aggregation

```
AggregationKey<Map<Genre, Long>> countsByGenreKey = AggregationKey.of( "countsByGenre" );
SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .aggregation( countsByGenreKey, f -> f.terms()
                .field( "genre", Genre.class )
                .minDocumentCount( 0 ) )
        .fetch( 20 );
Map<Genre, Long> countsByGenre = result.aggregation( countsByGenreKey );
```

This can also be used to omit entries with a document count that is too low to matter:

Example 15.230: Excluding the rarest terms from a `terms` aggregation

```
AggregationKey<Map<Genre, Long>> countsByGenreKey = AggregationKey.of( "countsByGenre" );
SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .aggregation( countsByGenreKey, f -> f.terms()
                .field( "genre", Genre.class )
                .minDocumentCount( 2 ) )
        .fetch( 20 );
Map<Genre, Long> countsByGenre = result.aggregation( countsByGenreKey );
```

Order of entries

By default, entries are returned in descending order of document count, i.e. the terms with the most matching documents appear first.

Several other orders are available.

You can order entries by ascending term value:

Example 15.231: Ordering entries by ascending value in a `terms` aggregation

```
AggregationKey<Map<Genre, Long>> countsByGenreKey = AggregationKey.of( "countsByGenre" );
SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .aggregation( countsByGenreKey, f -> f.terms()
                .field( "genre", Genre.class )
                .orderByTermAscending() )
        .fetch( 20 );
Map<Genre, Long> countsByGenre = result.aggregation( countsByGenreKey );
```

You can order entries by descending term value:

Example 15.232: Ordering entries by descending value in a `terms` aggregation

```
AggregationKey<Map<Genre, Long>> countsByGenreKey = AggregationKey.of( "countsByGenre" );
SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .aggregation( countsByGenreKey, f -> f.terms()
                .field( "genre", Genre.class )
                .orderByTermDescending() )
        .fetch( 20 );
Map<Genre, Long> countsByGenre = result.aggregation( countsByGenreKey );
```

Finally, you can order entries by ascending document count:

Example 15.233: Ordering entries by ascending count in a `terms` aggregation

```
AggregationKey<Map<Genre, Long>> countsByGenreKey = AggregationKey.of( "countsByGenre" );
SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .aggregation( countsByGenreKey, f -> f.terms()
                .field( "genre", Genre.class )
                .orderByCountAscending() )
        .fetch( 20 );
Map<Genre, Long> countsByGenre = result.aggregation( countsByGenreKey );
```

When ordering entries by ascending count in a `terms` aggregation, hit counts are approximate.

Aggregated value

By default, the aggregated value represents the number of documents that fall into the group of a particular term. With the `.value(..)` step in aggregation definition, it is now possible to set the aggregated value to something other than the document count. The `.value(..)` accepts any other aggregation, which will be applied to the documents within the aggregated group.

Example 15.234: Total price of books per category

```
AggregationKey<Map<Genre, Double>> sumByCategoryKey = AggregationKey.of( "sumByCategory" );
SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .aggregation(
                sumByCategoryKey, f -> f.terms()
                        .field( "genre", Genre.class ) ①
                        .value( f.sum().field( "price", Double.class ) ) ②
```

```
        )
        .fetch( 20 );
Map<Genre, Double> sumByPrice = result.aggregation( sumByCategoryKey );
```

① Define the path and type of the field whose values should be considered as terms for the aggregation.
② Define what the aggregated value should represent, e.g. the sum of all book prices within the genre.

Other options

- For fields in nested objects, all nested objects are considered by default, but that can be controlled explicitly with `.filter(…)`.

15.6.3. `range`: grouped by ranges of values for a field

The `range` aggregation returns a count of documents for given ranges of values of a given field.

In order to use aggregations based on the value of a given field, you need to mark the field as aggregable in the mapping.

This is not possible for full-text fields, in particular; see here for an explanation and some solutions.

The `range` aggregation is not available on geo-point fields.

Example 15.235: Counting hits grouped by range of values for a field

```
AggregationKey<Map<Range<Double>, Long>> countsByPriceKey = AggregationKey.of( "countsByPrice" );
SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .aggregation( countsByPriceKey, f -> f.range()
                .field( "price", Double.class ) ①
                .range( 0.0, 10.0 ) ②
                .range( 10.0, 20.0 )
                .range( 20.0, null ) ③
        )
        .fetch( 20 );
Map<Range<Double>, Long> countsByPrice = result.aggregation( countsByPriceKey );
```

① Define the path and type of the field whose values should be considered.
② Define the ranges to group hits into. The range can be passed directly as the lower bound (included) and upper bound (excluded). Other syntaxes exist to define different bound inclusion (see other examples below).
③ `null` means "to infinity".

Passing `Range` arguments

Instead of passing two arguments for each range (a lower and upper bound), you can pass a single argument of type `Range`.

Example 15.236: Counting hits grouped by range of values for a field — passing `Range` objects

```
AggregationKey<Map<Range<Double>, Long>> countsByPriceKey = AggregationKey.of( "countsByPrice" );
SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .aggregation( countsByPriceKey, f -> f.range()
                .field( "price", Double.class )
                .range( Range.canonical( 0.0, 10.0 ) ) ①
                .range( Range.between( 10.0, RangeBoundInclusion.INCLUDED,
                        20.0, RangeBoundInclusion.EXCLUDED ) ) ②
                .range( Range.atLeast( 20.0 ) ) ③
        )
        .fetch( 20 );
Map<Range<Double>, Long> countsByPrice = result.aggregation( countsByPriceKey );
```

① With `Range.of(Object, Object)`, the lower bound is included and the upper bound is excluded.
② `Range.of(Object, RangeBoundInclusion, Object, RangeBoundInclusion)` is more verbose, but allows setting the bound inclusion explicitly.

③ `Range` also offers multiple static methods to create ranges for a variety of use cases ("at least", "greater than", "at most", …).

With the Elasticsearch backend, due to a limitation of Elasticsearch itself, all ranges must have their lower bound included (or `null`) and their upper bound excluded (or `null`). Otherwise, an exception will be thrown.

If you need to exclude the lower bound, or to include the upper bound, replace that bound with the immediate next value instead. For example with integers, `.range( 0, 100 )` means "0 (included) to 100 (excluded)". Call `.range( 0, 101 )` to mean "0 (included) to 100 (included)", or `.range( 1, 100 )` to mean "0 (excluded) to 100 (excluded)".

It's also possible to pass a collection of `Range` objects, which is especially useful if ranges are defined dynamically (e.g. in a web interface):

Example 15.237: Counting hits grouped by range of values for a field — passing a collection of `Range` objects

```
List<Range<Double>> ranges =
/* ... */;

AggregationKey<Map<Range<Double>, Long>> countsByPriceKey = AggregationKey.of( "countsByPrice" );
SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .aggregation( countsByPriceKey, f -> f.range()
                .field( "price", Double.class )
                .ranges( ranges )
        )
        .fetch( 20 );
Map<Range<Double>, Long> countsByPrice = result.aggregation( countsByPriceKey );
```

Skipping conversion

By default, the bounds of ranges accepted by the `range` aggregation must have the same type as the entity property corresponding to the target field.

For example, if an entity property if of type `java.util.Date`, the corresponding field may be of type `java.time.Instant`; the values returned by the `terms` aggregation will have to be of type `java.util.Date` regardless.

This should generally be what you want, but if you ever need to bypass conversion and have unconverted values returned to you instead (of type `java.time.Instant` in the example above), you can do it this way:

Example 15.238: Counting hits grouped by range of values for a field, without converting field values

```
AggregationKey<Map<Range<Instant>, Long>> countsByPriceKey = AggregationKey.of( "countsByPrice" );
SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .aggregation( countsByPriceKey, f -> f.range()
                // Assuming "releaseDate" is of type "java.util.Date" or "java.sql.Date"
                .field( "releaseDate", Instant.class, ValueModel.INDEX )
                .range( null,
                        LocalDate.of( 1970, 1, 1 )
                                .atStartOfDay().toInstant( ZoneOffset.UTC ) )
                .range( LocalDate.of( 1970, 1, 1 )
                                .atStartOfDay().toInstant( ZoneOffset.UTC ),
                        LocalDate.of( 2000, 1, 1 )
                                .atStartOfDay().toInstant( ZoneOffset.UTC ) )
                .range( LocalDate.of( 2000, 1, 1 )
                                .atStartOfDay().toInstant( ZoneOffset.UTC ),
                        null )
        )
        .fetch( 20 );
Map<Range<Instant>, Long> countsByPrice = result.aggregation( countsByPriceKey );
```

See Type of arguments passed to the DSL for more information.

Aggregated value

By default, the aggregated value represents the number of documents that fall into particular, defined range . With the `.value(..)` step in aggregation definition, it is now possible to set the aggregated value to something other than the document count. The `.value(..)` accepts any other aggregation, which will be applied to the documents within the aggregated group.

Example 15.239: Total price of books per category

```
AggregationKey<Map<Range<Double>, Double>> avgRatingByPriceKey = AggregationKey.of( "avgRatingByPrice" );
SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .aggregation(
                avgRatingByPriceKey, f -> f.range()
                        .field( "price", Double.class ) ①
                        .range( 0.0, 10.0 )
                        .range( 10.0, 20.0 )
                        .range( 20.0, null )
                        .value( f.avg().field( "ratings", Double.class, ValueModel.RAW ) ) ②
        )
        .fetch( 20 );
Map<Range<Double>, Double> countsByPrice = result.aggregation( avgRatingByPriceKey );
```

① Define the path and type of the field whose value ranges for the aggregation.
② Define what the aggregated value should represent, e.g. the average rating of all books within the price range.

Other options

- For fields in nested objects, all nested objects are considered by default, but that can be controlled explicitly with `.filter(…)`.

15.6.4. Metric aggregations

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

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

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

Hibernate Search provides a set of most common metric aggregations such as `sum`, `min`, `max`, `count` and `avg`. These aggregations can be requested for fields of numerical or temporal types, or, in the case of `count`, at the document level (counting documents).

The fields that are targeted by the aggregation function must be declared aggregable.

Sum metric aggregation

The `sum` aggregation sums up the field values.

Example 15.240: Sum the prices of all science fiction books

```
AggregationKey<Double> sumPricesKey = AggregationKey.of( "sumPricesScienceFictionBooks" );
SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.match().field( "genre" ).matching( Genre.SCIENCE_FICTION ) )
        .aggregation( sumPricesKey, f -> f.sum().field( "price", Double.class ) ) ①
        .fetch( 20 );
Double sumPrices = result.aggregation( sumPricesKey );
```

① Define the target field path to which you want to apply the aggregation function and the expected returned type.

You can always use the field type for the expected returned value. Alternatively, as aggregations usually involve some mathematical computations, it is possible to request a `Double.class` result, which will return the aggregated value as the corresponding search backend computed it.

Min metric aggregation

The `min` aggregation provides the minimum value among the field values.

Example 15.241: Find the min release date among all the science fiction books

```
AggregationKey<Date> oldestReleaseKey = AggregationKey.of( "oldestRelease" );
SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.match().field( "genre" ).matching( Genre.SCIENCE_FICTION ) )
        .aggregation( oldestReleaseKey, f -> f.min().field( "releaseDate", Date.class ) ) ①
        .fetch( 20 );
Date oldestRelease = result.aggregation( oldestReleaseKey );
```

① Define the target field path to which you want to apply the aggregation function and the expected returned type.

Max metric aggregation

The `max` aggregation provides the maximum value among the field values.

Example 15.242: Find the max release date among all the science fiction books

```
AggregationKey<Date> mostRecentReleaseKey = AggregationKey.of( "mostRecentRelease" );
SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.match().field( "genre" ).matching( Genre.SCIENCE_FICTION ) )
        .aggregation( mostRecentReleaseKey, f -> f.max().field( "releaseDate", Date.class ) ) ①
        .fetch( 20 );
Date mostRecentRelease = result.aggregation( mostRecentReleaseKey );
```

① Define the target field path to which you want to apply the aggregation function and the expected returned type.

Count metric aggregation

The count aggregation allows counting either the documents that matched or field values.

The `count documents` aggregation counts the number of documents. While it is usually discouraged to use this aggregation at the root level, as the result would be equivalent to the count returned by the search results in `SearchResultTotal`, this aggregation can still be useful in defining aggregation values in other, more complex aggregations like `range` or `terms`.

Example 15.243: Count the number of the science fiction books

```
AggregationKey<Long> countBooksKey = AggregationKey.of( "countBooks" );
SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.match().field( "genre" ).matching( Genre.SCIENCE_FICTION ) )
        .aggregation( countBooksKey, f -> f.count() ①
                .documents() ) ②
        .fetch( 20 );
Long countPrices = result.aggregation( countBooksKey );
```

① Start building the counts aggregation by calling `.count()`. This aggregation always return a `Long.class` value.
② Request the document count.

The `count values` aggregation counts the number of non-empty field values. This aggregation mostly make sense when the aggregated field is multivalued. For single-valued fields this aggregation would result in the number of documents where the aggregated field is present.

Example 15.244: Count the number of the science fiction books with prices

```
AggregationKey<Long> countRatingsKey = AggregationKey.of( "countRatings" );
SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.match().field( "genre" ).matching( Genre.SCIENCE_FICTION ) )
        .aggregation( countRatingsKey, f -> f.count() ①
                .field( "ratings" ) ) ②
        .fetch( 20 );
```

```
Long countPrices = result.aggregation( countRatingsKey );
```

① Start building the counts aggregation by calling `.count()`. This aggregation always return a `Long.class` value.
② Specify the path of the field whose values to count.

Additionally, it is possible to count only the distinct field values.

Example 15.245: Count the number of all different price value among all the science fiction books

```
AggregationKey<Long> countDistinctPricesKey = AggregationKey.of( "countDistinctPrices" );
SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.match().field( "genre" ).matching( Genre.SCIENCE_FICTION ) )
        .aggregation( countDistinctPricesKey, f -> f.count() ①
                .field( "price" ) ②
                .distinct() ) ③
        .fetch( 20 );
Long countDistinctPrices = result.aggregation( countDistinctPricesKey );
```

① Start building the counts aggregation by calling `.count()`. This aggregation always return a `Long.class` value.
② Specify the path of the field whose values to count.
③ Request a count of distinct values (ignoring duplicates).

Avg metric aggregation

The `avg` aggregation calculates the average value of a given numeric or temporal field among the matched documents.

Example 15.246: Compute the average price of all science fiction books

```
AggregationKey<Double> avgPricesKey = AggregationKey.of( "avgPrices" );
SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.match().field( "genre" ).matching( Genre.SCIENCE_FICTION ) )
        .aggregation( avgPricesKey, f -> f.avg().field( "price", Double.class ) ) ①
        .fetch( 20 );
Double avgPrices = result.aggregation( avgPricesKey );
```

① Define the target field path to which you want to apply the aggregation function and the expected returned type.

In the case of the `avg()` aggregation, the result may have some decimal values, even if the field type is represented by one of the integer number types. In this case, if you want to return decimals, provide `Double.class` for the expected returned type.

15.6.5. `composite`: combine aggregations

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

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

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

Basics

The `composite` aggregation applies multiple aggregations and combines their results, either as a `List<?>`/`Object[]` or as a single object generated using a custom transformer.

While composite aggregations can be defined at the root level, they are most useful as a container to collect multiple values from terms and range aggregations.

Constructing composite aggregations is quite similar to composite projections.

```
record PriceAggregation(Double avg, Double min, Double max) {
}

AggregationKey<PriceAggregation> avgPricesKey = AggregationKey.of( "aggregations" );
SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .aggregation( avgPricesKey, f -> f.composite() ①
                .from(
                        f.avg().field( "price", Double.class ), ②
                        f.min().field( "price", Double.class ),
                        f.max().field( "price", Double.class )
                ).as( PriceAggregation::new ) ) ③
        .fetch( 20 );
PriceAggregation aggregations = result.aggregation( avgPricesKey ); ④
```

① Call `.composite()`.

② Define the inner aggregations on the `price` field. While this example uses the same field it is possible to use different fields within the same composite aggregation.

③ Define the result of the composite aggregation as the result of calling the constructor of a custom object, `PriceAggregation`. The constructor of `PriceAggregation` will be called for the aggregated results and resulting object can be than retrieved from the fetched search results.

④ Retrieving the `PriceAggregation` as aggregation results.

Composing more than 3 inner aggregations

If you require more than 3 aggregations as arguments to `from(…)`, then the transform function will have to take a `List<?>` as an argument, and will be set using `asList(…)` instead of `as(...)`:

Example 15.248: Returning custom object created from multiple aggregation values with `.composite().from(…).asList(…)`

```
record BookAggregation(Double avg, Double min, Double max, Long ratingCount) {
    BookAggregation(List<?> list) {
        this( (Double) list.get( 0 ),
                (Double) list.get( 1 ),
                (Double) list.get( 2 ),
                (Long) list.get( 3 ) );
    }
}

AggregationKey<BookAggregation> aggKey = AggregationKey.of( "aggregations" );
SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .aggregation( aggKey, f -> f.composite() ①
                .from(
                        f.avg().field( "price", Double.class ), ②
                        f.min().field( "price", Double.class ),
                        f.max().field( "price", Double.class ),
                        f.count().field( "ratings" )
                ).asList( BookAggregation::new ) ) ③
        .fetch( 20 );
BookAggregation aggregations = result.aggregation( aggKey ); ④
```

① Call `.composite()`.

② Define the inner aggregations on the `price` and `ratings` fields.

③ Define the result of the aggregation as the result of calling a `list → obj` function. The lambda will take elements of the list (the results of aggregations defined above, in that same order). For convenience an alternative constructor taking the list is defined.

④ Retrieving the `BookAggregation` as aggregation results.

Aggregating to a `List<?>` or `Object[]`

If you don't mind receiving the result of inner aggregations as a `List<?>`, you can do without the transformer by simply calling `asList()`:

Example 15.249: Returning a `List` of aggregated values with `.composite().add(…).asList()`

```
AggregationKey<List<?>> aggKey = AggregationKey.of( "aggregations" );
SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .aggregation( aggKey, f -> f.composite() ①
                .from(
                        f.avg().field( "price", Double.class ), ②
                        f.min().field( "price", Double.class ),
                        f.max().field( "price", Double.class ),
                        f.count().field( "ratings" )
                ).asList() ) ③
        .fetch( 20 );
List<?> aggregations = result.aggregation( aggKey ); ④
```

① Call `.composite()`.

② Define the inner aggregations on the `price` and `ratings` fields.

③ Define the result of the aggregations as a list, meaning the aggregated result will be `List` instance with the value of the average `price` of the matched documents at index `0`, the value of the minimum `price` of the matched documents at index `1`, the value of the maximum `price` of the matched documents at index `2`, and the total number of `ratings` field values of the matched documents at index `3`.

④ Retrieving the `List<?>` as aggregation results.

Similarly, to get the result of inner aggregations as an array (`Object[]`), you can do without the transformer by calling `asArray()`:

Example 15.250: Returning an array of aggregated values with `.composite(…).add(…).asArray()`

```
AggregationKey<Object[]> aggKey = AggregationKey.of( "aggregations" );
SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .aggregation( aggKey, f -> f.composite() ①
                .from(
                        f.avg().field( "price", Double.class ), ②
                        f.min().field( "price", Double.class ),
                        f.max().field( "price", Double.class ),
                        f.count().field( "ratings" )
                ).asArray() ) ③
        .fetch( 20 );
Object[] aggregations = result.aggregation( aggKey ); ④
```

① Call `.composite()`.

② Define the inner aggregations on the `price` and `ratings` fields.

③ Define the result of the aggregations as an array, meaning the aggregated result will be `Object[]` instance with the value of the average `price` of the matched documents at index `0`, the value of the minimum `price` of the matched documents at index `1`, the value of the maximum `price` of the matched documents at index `2`, and the total number of `ratings` field values of the matched documents at index `3`.

④ Retrieving the `Object[]` as aggregation results.

Alternatively, to get the result as a `List<?>`, you can use the shorter variant of `.composite(…)` that directly takes aggregations as arguments:

Example 15.251: Returning a `List` of aggregated values with `.composite(…)`

```
AggregationKey<List<?>> aggKey = AggregationKey.of( "aggregations" );
SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .aggregation( aggKey, f -> f.composite( ①
                f.avg().field( "price", Double.class ), ②
                f.min().field( "price", Double.class ),
                f.max().field( "price", Double.class ),
                f.count().field( "ratings" )
        ) ) ③
        .fetch( 20 );
List<?> aggregations = result.aggregation( aggKey ); ④
```

① Call `.composite(…)` and pass all of the required aggregations to it.
② Define the inner aggregations on the `price` and `ratings` fields.
③ No extra `.as()`/`.asList()`/`.asArray()` method calls required.
④ Retrieving the `List<?>` as aggregation results.

15.6.6. `withParameters`: create aggregations using query parameters

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

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

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

The `withParameters` aggregation allows building aggregations using query parameters.

This type of aggregation requires a function that accepts query parameters and returns an aggregation. That function will get called at query building time.

Example 15.252: Creating an aggregation with query parameters

```
AggregationKey<Map<Range<Double>, Long>> countsByPriceKey = AggregationKey.of( "countsByPrice" );
SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .aggregation( countsByPriceKey, f -> f.withParameters( params -> f.range() ①
                .field( "price", Double.class )
                .range( params.get( "bound0", Double.class ), params.get( "bound1", Double.class ) ) ②
                .range( params.get( "bound1", Double.class ), params.get( "bound2", Double.class ) )
                .range( params.get( "bound2", Double.class ), params.get( "bound3", Double.class ) )
        ) )
        .param( "bound0", 0.0 ) ③
        .param( "bound1", 10.0 )
        .param( "bound2", 20.0 )
        .param( "bound3", null )
        .fetch( 20 );
Map<Range<Double>, Long> countsByPrice = result.aggregation( countsByPriceKey );
```

① Start creating the `.withParameters()` aggregation.
② Access the query parameters of `Double` type, defining range bounds, when constructing the aggregation.
③ Set parameters required by the aggregation at the query level.

15.6.7. Backend-specific extensions

By calling `.extension(…)` while building a query, it is possible to access backend-specific aggregations.

As their name suggests, backend-specific aggregations are not portable from one backend technology to the other.

Elasticsearch: `fromJson`

`.fromJson(…)` turns JSON representing an Elasticsearch aggregation into a Hibernate Search aggregation.

This feature requires to directly manipulate JSON in application code.

The syntax of this JSON may change:

- when you upgrade the underlying Elasticsearch cluster to the next version;
- when you upgrade Hibernate Search to the next version, even for a bugfix (micro) release.

If this happens, you will need to change application code to deal with the changes.

Example 15.253: Defining a native Elasticsearch JSON aggregation as a `JsonObject`

```
JsonObject jsonObject =
/* ... */;
AggregationKey<JsonObject> countsByPriceHistogramKey = AggregationKey.of( "countsByPriceHistogram" );
SearchResult<Book> result = searchSession.search( Book.class )
        .extension( ElasticsearchExtension.get() )
        .where( f -> f.matchAll() )
        .aggregation( countsByPriceHistogramKey, f -> f.fromJson( jsonObject ) )
        .fetch( 20 );
JsonObject countsByPriceHistogram = result.aggregation( countsByPriceHistogramKey ); ①
```

① The aggregation result is a `JsonObject`.

Example 15.254: Defining a native Elasticsearch JSON aggregation as a JSON-formatted string

```
AggregationKey<JsonObject> countsByPriceHistogramKey = AggregationKey.of( "countsByPriceHistogram" );
SearchResult<Book> result = searchSession.search( Book.class )
        .extension( ElasticsearchExtension.get() )
        .where( f -> f.matchAll() )
        .aggregation( countsByPriceHistogramKey, f -> f.fromJson( "{"
                + "    \"histogram\": {"
                + "        \"field\": \"price\","
                + "        \"interval\": 10"
                + "    }"
                + "}" ) )
        .fetch( 20 );
JsonObject countsByPriceHistogram = result.aggregation( countsByPriceHistogramKey ); ①
```

① The aggregation result is a `JsonObject`.

15.6.8. Options common to multiple aggregation types

Filter for fields in nested objects

When the aggregation field is located in a nested object, by default all nested objects will be considered for the aggregation, and the document will be counted once for each value found in any nested object.

It is possible to filter the nested documents whose values will be considered for the aggregation using one of the `filter(…)` methods.

Below is an example with the range aggregation: the result of the aggregation is a count of books for each price range, with only the price of "paperback" editions being taken into account; the price of e-book editions, for example, is ignored.

Example 15.255: Counting hits grouped by range of values for a field, using a filter for nested objects

```
AggregationKey<Map<Range<Double>, Long>> countsByPriceKey = AggregationKey.of( "countsByPrice" );
SearchResult<Book> result = searchSession.search( Book.class )
        .where( f -> f.matchAll() )
        .aggregation( countsByPriceKey, f -> f.range()
                .field( "editions.price", Double.class )
                .range( 0.0, 10.0 )
                .range( 10.0, 20.0 )
                .range( 20.0, null )
                .filter( pf -> pf.match().field( "editions.label" ).matching( "paperback" ) )
        )
        .fetch( 20 );
Map<Range<Double>, Long> countsByPrice = result.aggregation( countsByPriceKey );
```

15.7. Field types and compatibility

15.7.1. Type of arguments passed to the DSL

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

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

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

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

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

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

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

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

Example 15.256: Transparent conversion of DSL parameters

```
List<AuthenticationEvent> result = searchSession.search( AuthenticationEvent.class )
        .where( f -> f.match().field( "outcome" )
                .matching( AuthenticationOutcome.INVALID_PASSWORD ) )
        .fetchHits( 20 );
```

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

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

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

Example 15.257: Disabling the DSL converter

```
List<AuthenticationEvent> result = searchSession.search( AuthenticationEvent.class )
        .where( f -> f.match().field( "outcome" )
                .matching( "Invalid password", ValueModel.INDEX ) )
        .fetchHits( 20 );
```

All methods that apply DSL converters offer a variant that accepts a `ValueModel` parameter: `matching`, `between`, `atLeast`, `atMost`, `greaterThan`, `lessThan`, `range`, …

In some cases, it may be helpful to pass string values to these DSL steps. `ValueModel.STRING` can be used to address that. By default, the string format should be compatible with the parsing logic defined in Property types with built-in value bridges, alternatively see how it can be customized with bridges.

Example 15.258: Using the `STRING` DSL converter to work with string arguments

```
List<AuthenticationEvent> result = searchSession.search( AuthenticationEvent.class )
        .where( f -> f.match().field( "time" )
```

```
                .matching( "2002-02-20T20:02:22", ValueModel.STRING ) )
        .fetchHits( 20 );
```

There is also a possibility to use "raw" types that search backends operate with by passing `ValueModel.RAW` to the DSL step.

Raw types are both backend and implementation specific. Use `ValueModel.RAW` cautiously and be aware that the format or types themselves may change in the future.

Currently, the Elasticsearch backend uses `String` representation of raw types, while the Lucene backend uses a variety of types depending on how a particular filed is stored in the index. Inspect the corresponding implementation of the `LuceneFieldCodec` to identify the type.

Example 15.259: Using the `RAW` DSL converter to work with raw arguments

```
Object rawDateTimeValue = // ... ①
List<AuthenticationEvent> result = searchSession.search( AuthenticationEvent.class )
        .where( f -> f.match().field( "time" )
                .matching( rawDateTimeValue, ValueModel.RAW ) )
        .fetchHits( 20 );
```

① Keep in mind that raw value is backend specific.

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

See Type bridge or Property bridge for more information.

15.7.2. Type of projected values

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

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

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

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

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

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

Example 15.260: Transparent conversion of projections

```
List<OrderStatus> result = searchSession.search( Order.class )
        .select( f -> f.field( "status", OrderStatus.class ) )
        .where( f -> f.matchAll() )
        .fetchHits( 20 );
```

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

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

Example 15.261: Disabling the projection converter

```
List<String> result = searchSession.search( Order.class )
        .select( f -> f.field( "status", String.class, ValueModel.INDEX ) )
        .where( f -> f.matchAll() )
        .fetchHits( 20 );
```

There is also a possibility to request "raw" types that search backends operate with by passing `ValueModel.RAW` to the projection DSL step.

Raw types are both backend and implementation specific. Use `ValueModel.RAW` cautiously and be aware that the format or types themselves may change in the future.

Currently, the Elasticsearch backend uses `String` representation of raw types, while the Lucene backend uses a variety of types depending on how a particular filed is stored in the index. Inspect the corresponding implementation of the `LuceneFieldCodec` to identify the type.

Example 15.262: Using the `RAW` projection converter to work with raw projections

```
Class<String> rawProjectionType = // ... ①
List<?> result = searchSession.search( Order.class )
        .select( f -> f.field( "status", rawProjectionType, ValueModel.RAW ) )
        .where( f -> f.matchAll() )
        .fetchHits( 20 );
```

① Keep in mind that raw value is backend specific.

Projection converters must be configured explicitly in custom bridges.

See Value bridge, Property bridge or Type bridge for more information.

15.7.3. Targeting multiple fields

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

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

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

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

However, if you're interested in the details, Hibernate Search is a bit more flexible than that.

There are three different constraints when it comes to field compatibility:

1. The fields must be "encoded" in a compatible way. This means the backend must use the same representation for the two fields, for example they are both `Integer`, or they are both `BigDecimal` with the same decimal scale, or they are both `LocalDate` with the same date format, etc.
2. The fields must have a compatible DSL converter (for predicates and sorts) or projection converter (for projections).
3. For full-text predicates, the fields must have a compatible analyzer.

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

Incompatible codec

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

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

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

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

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

Incompatible DSL converters

Incompatible DSL converters are only a problem when you need to pass an argument to the DSL in certain methods: `matching(Object)` /`between(Object)`/`atLeast(Object)`/`greaterThan(Object)`/etc. in the predicate DSL, `missing().use(Object) in the sort DSL, `range(Object, Object)` in the aggregation DSL, …

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

Incompatible projection converters

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

Incompatible analyzer

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

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

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

15.8. Field paths

15.8.1. Absolute field paths

By default, field paths passed to the Search DSL are interpreted as absolute, i.e. relative to the index root.

The components of the paths are separated by a dot (`.`).

The following examples use the predicate DSL, but all information in this section applies to other search DSLs as well: sort DSL, projection DSL, aggregation DSL, …

Example 15.263: Targeting a field using absolute paths

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.match().field( "title" ) ①
                .matching( "robot" ) )
        .fetchHits( 20 );
```

① Fields declared at the root of the index can simply be referenced by their name.

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.match().field( "writers.firstName" ) ①
                .matching( "isaac" ) )
        .fetchHits( 20 );
```

① Fields declared in object fields (created by `@IndexedEmbedded`, for example) must be referenced by their absolute path: the

absolute path of the object field, followed by a dot, followed by the name of the targeted field.

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.nested( "writers" )
                .add( f.match().field( "writers.firstName" ) ①
                        .matching( "isaac" ) )
                .add( f.match().field( "writers.lastName" )
                        .matching( "asimov" ) )
        )
        .fetchHits( 20 );
```

① Even within a `nested` predicate, fields referenced in inner predicates must be referenced by their absolute path.

The only exception is named predicates registered on object fields: the factory used to build those predicates interprets field paths as relative to that object field **by default**.

15.8.2. Relative field paths

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

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

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

In some cases, one may want to pass relative paths instead. This can be useful when calling reusable methods that can apply the same predicate on different object fields that have same structure (same subfields). By calling the `withRoot(String)` method on a factory, you can create a new factory which interprets paths as relative to the object field passed as argument to the method.

Example 15.264: Targeting a field using relative paths

```
List<Book> hits = searchSession.search( Book.class )
        .where( f -> f.or()
                .add( f.nested( "writers" )
                        .add( matchFirstAndLastName( ①
                                f.withRoot( "writers" ), ②
                                "bob", "kane" ) ) )
                .add( f.nested( "artists" )
                        .add( matchFirstAndLastName( ③
                                f.withRoot( "artists" ), ④
                                "bill", "finger" ) ) ) )
        .fetchHits( 20 );
```

① Call a reusable method to apply a predicate to the name of the book's writers.
② Pass to that method a factory whose root will be the object field `writers`.
③ Call a reusable method to apply a predicate to the name of the book's artists.
④ Pass to that method a factory whose root will be the object field `artists`.

```
private SearchPredicate matchFirstAndLastName(SearchPredicateFactory f,
        String firstName, String lastName) {
    return f.and(
            f.match().field( "firstName" ) ①
                    .matching( firstName ),
            f.match().field( "lastName" )
                    .matching( lastName )
    )
            .toPredicate();
}
```

① When manipulating factories created with `withRoot`, paths are interpreted as relative. Here `firstName` will be understood as either `writers.firstName` or `artists.firstName`, depending on the factory that was passed to this method.

When building native constructs (for example Lucene Queries), you will need to deal with absolute paths, even if the

> factory accepts relative paths.
>
> To convert a relative path to an absolute path, use the factory's `toAbsolutePath(String)` method.

Chapter 16. Explicit backend/index operations

16.1. Applying configured analyzers/normalizers to a string

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

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

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

Hibernate Search provides an API that applies an analyzer/normalizer to a given string. This can be useful to test how these analyzers/normalizers work.

Example 16.1: Inspecting tokens produced by a configured analyzer.

```
SearchMapping mapping = /* ... */ ①
IndexManager indexManager = mapping.indexManager( "Book" ); ②

List<? extends AnalysisToken> tokens = indexManager.analyze( ③
        "my-analyzer", ④
        "The quick brown fox jumps right over the little lazy dog" ⑤
);
for ( AnalysisToken token : tokens ) { ⑥
    String term = token.term();
    int startOffset = token.startOffset();
    int endOffset = token.endOffset();
    // ...
}
```

① Retrieve the `SearchMapping`.
② Retrieve the `IndexManager`.
③ Perform the analysis.
④ Pass the name of a configured analyzer.
⑤ Pass the text on which the analysis should be performed.
⑥ Inspect the tokens produced by the analysis.

Example 16.2: Inspecting tokens produced by a configured normalizer.

```
SearchMapping mapping = /* ... */ ①
IndexManager indexManager = mapping.indexManager( "Book" ); ②

AnalysisToken normalizedToken = indexManager.normalize( ③
        "my-normalizer", ④
        "The quick brown fox jumps right over the little lazy dog" ⑤
);
String term = normalizedToken.term(); ⑥
// ...
```

① Retrieve the `SearchMapping`.
② Retrieve the `IndexManager`.
③ Perform the normalization.
④ Pass the name of a configured normalizer.
⑤ Pass the text to which the normalizer should be applied.
⑥ Inspect the token produced by the normalizer.

There are also async versions of the methods to perform analysis/normalization: `analyzeAsync(..)`/`normalizeAsync(..)`.

16.2. Explicitly altering a whole index

Some index operations are not about a specific entity/document, but rather about a large number of documents, possibly all of them. This includes, for example, purging the index to remove all of its content.

The operations are accessed through the `SearchWorkspace` interface, and executed immediately (**outside** the context of a `SearchSession`, Hibernate ORM session or transaction).

The `SearchWorkspace` can be retrieved from the `SearchMapping`, and can target one, several or all indexes:

Example 16.3: Retrieving a `SearchWorkspace` from the `SearchMapping`

```
SearchMapping searchMapping = /* ... */ ①
SearchWorkspace allEntitiesWorkspace = searchMapping.scope( Object.class ).workspace(); ②
SearchWorkspace bookWorkspace = searchMapping.scope( Book.class ).workspace(); ③
SearchWorkspace bookAndAuthorWorkspace = searchMapping.scope( Arrays.asList( Book.class, Author.class ) )
        .workspace(); ④
```

① Retrieve the `SearchMapping`.
② Get a workspace targeting all indexes.
③ Get a workspace targeting the index mapped to the `Book` entity type.
④ Get a workspace targeting the indexes mapped to the `Book` and `Author` entity types.

Alternatively, for convenience, the `SearchWorkspace` can be retrieved from the `SearchSession`:

Example 16.4: Retrieving a `SearchWorkspace` from the `SearchSession`

```
SearchMapping searchMapping = /* ... */ ①
SearchWorkspace allEntitiesWorkspace = searchMapping.scope( Object.class ).workspace(); ②
SearchWorkspace bookWorkspace = searchMapping.scope( Book.class ).workspace(); ③
SearchWorkspace bookAndAuthorWorkspace = searchMapping.scope( Arrays.asList( Book.class, Author.class ) )
        .workspace(); ④
```

① Retrieve the `SearchSession`.
② Get a workspace targeting all indexes.
③ Get a workspace targeting the index mapped to the `Book` entity type.
④ Get a workspace targeting the indexes mapped to the `Book` and `Author` entity types.

The `SearchWorkspace` exposes various large-scale operations that can be applied to an index or a set of indexes. These operations are triggered as soon as they are requested, without waiting for the `SearchSession` to be closed or the Hibernate ORM transaction to be committed.

This interface offers the following methods:

`purge()`
: Delete all documents from indexes targeted by this workspace.

  With multi-tenancy enabled, only documents of the current tenant will be removed: the tenant of the session from which this workspace originated.

`purgeAsync()`
: Asynchronous version of `purge()` returning a `CompletionStage`.

`purge(Set<String> routingKeys)`
: Delete documents from indexes targeted by this workspace that were indexed with any of the given routing keys.

  With multi-tenancy enabled, only documents of the current tenant will be removed: the tenant of the session from which this workspace originated.

`purgeAsync(Set<String> routingKeys)`
: Asynchronous version of `purge(Set<String>)` returning a `CompletionStage`.

`flush()`
: Flush to disk the changes to indexes that have not been committed yet. In the case of backends with a transaction log

(Elasticsearch), also apply operations from the transaction log that were not applied yet.

This is generally not useful as Hibernate Search commits changes automatically. See Commit and refresh for more information.

`flushAsync()`

Asynchronous version of `flush()` returning a `CompletionStage`.

`refresh()`

Refresh the indexes so that all changes executed so far will be visible in search queries.

This is generally not useful as indexes are refreshed automatically. See Commit and refresh for more information.

`refreshAsync()`

Asynchronous version of `refresh()` returning a `CompletionStage`.

`mergeSegments()`

Merge each index targeted by this workspace into a single segment. This operation does not always improve performance: see Merging segments and performance.

`mergeSegmentsAsync()`

Asynchronous version of `mergeSegments()` returning a `CompletionStage`. This operation does not always improve performance: see Merging segments and performance.

Merging segments and performance

The merge-segments operation may affect performance positively as well as negatively.

This operation will regroup all index data into a single, huge segment (a file). This may speed up search at first, but as documents are deleted, this huge segment will begin to fill with "holes" which have to be handled as special cases during search, degrading performance.

Elasticsearch/Lucene do address this by rebuilding the segment at some point, but only once a certain ratio of deleted documents is reached. If all documents are in a single, huge segment, this ratio is less likely to be reached, and the index performance will continue to degrade for a long time.

There are, however, two situations in which merging segments may help:

1. No deletions or document updates are expected for an extended period of time.
2. Most, or all documents have just been removed from the index, leading to segments consisting mostly of deleted documents. In that case, it makes sense to regroup the few remaining documents into a single segment, though Elasticsearch/Lucene will probably do it automatically.

Below is an example using a `SearchWorkspace` to purge several indexes.

Example 16.5: Purging indexes using a `SearchWorkspace`

```
SearchSession searchSession = /* ... */ ①
SearchWorkspace workspace = searchSession.workspace( Book.class, Author.class ); ②
workspace.purge(); ③
```

① Retrieve the `SearchSession`.
② Get a workspace targeting the indexes mapped to the `Book` and `Author` entity types.
③ Trigger a purge. This method is synchronous and will only return after the purge is complete, but an asynchronous method, `purgeAsync`, is also available.

16.3. Lucene-specific explicit backend/index operations

16.3.1. Retrieving analyzers and normalizers through the Lucene-specific `Backend`

Lucene analyzers and normalizers defined in Hibernate Search can be retrieved from the Lucene backend.

Example 16.6: Retrieving the Lucene analyzers by name from the backend

```
SearchMapping mapping = /* ... */ ①
Backend backend = mapping.backend(); ②
```

```
LuceneBackend luceneBackend = backend.unwrap( LuceneBackend.class ); ③
Optional<? extends Analyzer> analyzer = luceneBackend.analyzer( "english" ); ④
Optional<? extends Analyzer> normalizer = luceneBackend.normalizer( "isbn" ); ⑤
```

① Retrieve the `SearchMapping`.
② Retrieve the `Backend`.
③ Narrow down the backend to the `LuceneBackend` type.
④ Get an analyzer by name. The method returns an `Optional`, which is empty if the analyzer does not exist. The analyzer must have been defined in Hibernate Search, otherwise it won't exist.
⑤ Get a normalizer by name. The method returns an `Optional`, which is empty if the normalizer does not exist. The normalizer must have been defined in Hibernate Search, otherwise it won't exist.

Alternatively, you can also retrieve the (composite) analyzers for a whole index. These analyzers behave differently for each field, delegating to the analyzer configured in the mapping for each field.

Example 16.7: Retrieving the Lucene analyzers for a whole index

```
SearchMapping mapping = /* ... */ ①
IndexManager indexManager = mapping.indexManager( "Book" ); ②
LuceneIndexManager luceneIndexManager = indexManager.unwrap( LuceneIndexManager.class ); ③
Analyzer indexingAnalyzer = luceneIndexManager.indexingAnalyzer(); ④
Analyzer searchAnalyzer = luceneIndexManager.searchAnalyzer(); ⑤
```

① Retrieve the `SearchMapping`.
② Retrieve the `IndexManager`.
③ Narrow down the index manager to the `LuceneIndexManager` type.
④ Get the indexing analyzer. This is the analyzer used when indexing documents. It ignores the search analyzer in particular.
⑤ Get the search analyzer. This is the analyzer used when building search queries through the Search DSL. On contrary to the indexing analyzer, it takes into account the search analyzer.

16.3.2. Retrieving the Lucene's index size

The size of a Lucene index can be retrieved from the `LuceneIndexManager`.

Example 16.8: Retrieving the index size from a Lucene index manager

```
SearchMapping mapping = /* ... */ ①
IndexManager indexManager = mapping.indexManager( "Book" ); ②
LuceneIndexManager luceneIndexManager = indexManager.unwrap( LuceneIndexManager.class ); ③
long size = luceneIndexManager.computeSizeInBytes(); ④
luceneIndexManager.computeSizeInBytesAsync() ⑤
        .thenAccept( sizeInBytes -> {
            // ...
        } );
```

① Retrieve the `SearchMapping`.
② Retrieve the `IndexManager`.
③ Narrow down the index manager to the `LuceneIndexManager` type.
④ Compute the index size and get the result.
⑤ An asynchronous version of the method is also available.

16.3.3. Retrieving a Lucene `IndexReader`

The low-level `IndexReader` can be retrieved from the `LuceneIndexScope`.

Example 16.9: Retrieving the index reader from a Lucene index scope

```
SearchMapping mapping = /* ... */ ①
LuceneIndexScope indexScope = mapping
        .scope( Book.class ).extension( LuceneExtension.get() ); ②
```

```
try ( IndexReader indexReader = indexScope.openIndexReader() ) { ③
    // work with the low-level index reader:
    numDocs = indexReader.numDocs();
}
```

① Retrieve the `SearchMapping`.
② Retrieve the `LuceneIndexScope` extending the search scope with the `LuceneExtension`.
③ Open an instance of `IndexReader`. Optionally it is possible to provide the routing keys to target only some shards of the indexes using the method `openIndexReader(Set<String>)`. The IndexReader **must be closed** after use.

Even if multi-tenancy is enabled, the returned reader exposes documents of **all** tenants.

16.4. Elasticsearch-specific explicit backend/index operations

16.4.1. Retrieving the REST client

When writing complex applications with advanced requirements, it may be necessary from time to time to send requests to the Elasticsearch cluster directly, in particular if Hibernate Search does not support this kind of requests out of the box.

To that end, you can retrieve the Elasticsearch backend, then get access the Elasticsearch client used by Hibernate Search internally. See below for an example.

Example 16.10: Accessing the low-level REST client

```
SearchMapping mapping = /* ... */ ①
Backend backend = mapping.backend(); ②
ElasticsearchBackend elasticsearchBackend = backend.unwrap( ElasticsearchBackend.class ); ③
RestClient client = elasticsearchBackend.client( RestClient.class ); ④
```

① Retrieve the `SearchMapping`.
② Retrieve the `Backend`.
③ Narrow down the backend to the `ElasticsearchBackend` type.
④ Get the client, passing the expected type of the client as an argument.

The client itself is not part of the Hibernate Search API, but of the official Elasticsearch REST client API.

Hibernate Search may one day switch to another client with a different Java type, without prior notice. If that happens, the snippet of code above will throw an exception.

Chapter 17. Lucene backend

17.1. Basic configuration

All configuration properties of the Lucene backend are optional, but the defaults might not suit everyone. In particular, you might want to set the location of your indexes in the filesystem.

Other configuration properties are mentioned in the relevant parts of this documentation. You can find a full reference of available properties in the Lucene backend configuration properties appendix.

17.2. Index storage (`Directory`)

The component responsible for index storage in Lucene is the `org.apache.lucene.store.Directory`. The implementation of the directory determines where the index will be stored: on the filesystem, in the JVM's heap, …

By default, the Lucene backend stores the indexes on the filesystem, in the JVM's working directory.

The type of directory can be configured as follows:

```
# To configure the defaults for all indexes:
hibernate.search.backend.directory.type = local-filesystem
# To configure a specific index:
hibernate.search.backend.indexes.<index-name>.directory.type = local-filesystem
```

The following directory types are available:

- `local-filesystem`: Store the index on the local filesystem. See Local filesystem storage for details and configuration options.
- `local-heap`: Store the index in the local JVM heap. **Local heap directories and all contained indexes are lost when the JVM shuts down.** See Local heap storage for details and configuration options.

17.2.1. Local filesystem storage

The `local-filesystem` directory type will store each index in a subdirectory of a configured filesystem directory.

Local filesystem directories really are designed to be **local** to one server and one application.

In particular, they should not be shared between multiple Hibernate Search instances. Even if network shares allow to share the raw content of indexes, using the same index files from multiple Hibernate Search would require more than that: non-exclusive locking, routing of write requests from one node to another, … These additional features are simply not available on `local-filesystem` directories.

If you need to share indexes between multiple Hibernate Search instances, the Elasticsearch backend will be a better choice. Refer to Architecture for more information.

Index location

Each index is assigned a subdirectory under a root directory.

By default, the root directory is the JVM's working directory. It can be configured as follows:

```
# To configure the defaults for all indexes:
hibernate.search.backend.directory.root = /path/to/my/root
# To configure a specific index:
hibernate.search.backend.indexes.<index-name>.directory.root = /path/to/my/root
```

For example, with the configuration above, an entity type named `Order` will be indexed in directory `/path/to/my/root/Order/`. If that entity is explicitly assigned the index name `orders` (see `@Indexed(index = …)` in Entity/index mapping), it will instead be indexed in directory `/path/to/my/root/orders/`.

Filesystem access strategy

The default strategy for accessing the filesystem is determined automatically based on the operating system and architecture. It should work well in most situations.

For situations where a different filesystem access strategy is needed, Hibernate Search exposes a configuration property:

```
# To configure the defaults for all indexes:
hibernate.search.backend.directory.filesystem_access.strategy = auto
# To configure a specific index:
hibernate.search.backend.indexes.<index-name>.directory.filesystem_access.strategy = auto
```

Allowed values are:

- `auto`: lets Lucene select the most appropriate implementation based on the operating system and architecture. This is the default value for the property.
- `mmap`: uses `mmap` for reading, and `FSDirectory.FSIndexOutput` for writing. See `org.apache.lucene.store.MMapDirectory`.
- `nio`: uses `java.nio.channels.FileChannel`'s positional read for concurrent reading, and `FSDirectory.FSIndexOutput` for writing. See `org.apache.lucene.store.NIOFSDirectory`.

Make sure to refer to Javadocs of these `Directory` implementations before changing this setting. Implementations offering better performance also bring issues of their own.

Other configuration options

The `local-filesystem` directory also allows configuring a locking strategy.

17.2.2. Local heap storage

The `local-heap` directory type will store indexes in the local JVM's heap.

As a result, indexes contained in a `local-heap` directory are **lost when the JVM shuts down**.

This directory type is only provided for use in **testing configurations** with **small indexes** and **low concurrency**, where it could slightly improve performance. In setups requiring larger indexes and/or high concurrency, a filesystem-based directory will achieve better performance.

The `local-heap` directory does not offer any specific option beyond the locking strategy.

17.2.3. Locking strategy

In order to write to an index, Lucene needs to acquire a lock to ensure no other application instance writes to the same index concurrently. Each directory type comes with a default locking strategy that should work well enough in most situations.

For those (very) rare situations where a different locking strategy is needed, Hibernate Search exposes a configuration property:

```
# To configure the defaults for all indexes:
hibernate.search.backend.directory.locking.strategy = native-filesystem
# To configure a specific index:
hibernate.search.backend.indexes.<index-name>.directory.locking.strategy = native-filesystem
```

The following strategies are available:

- `simple-filesystem`: Locks the index by creating a marker file and checking it before write operations. This implementation is very simple and based Java's File API. If for some reason an application ends abruptly, the marker file will stay on the filesystem and will need to be removed manually.

  This strategy is only available for filesystem-based directories.

  See `org.apache.lucene.store.SimpleFSLockFactory`.

- `native-filesystem`: Similarly to `simple-filesystem`, locks the index by creating a marker file, but using native OS file locks instead of Java's File API, so that locks will be cleaned up if the application ends abruptly.

  This is the default strategy for the `local-filesystem` directory type.

  This implementation has known problems with NFS: it should be avoided on network shares.

  This strategy is only available for filesystem-based directories.

  See `org.apache.lucene.store.NativeFSLockFactory`.

- `single-instance`: Locks using a Java object held in the JVM's heap. Since the lock is only accessible by the same JVM, this strategy will only work properly when it is known that only a single application will ever try to access the indexes.

This is the default strategy for the `local-heap` directory type.

See `org.apache.lucene.store.SingleInstanceLockFactory`.

- `none`: Does not use any lock. Concurrent writes from another application will result in index corruption. Test your application carefully and make sure you know what it means.

  See `org.apache.lucene.store.NoLockFactory`.

17.3. Sharding

17.3.1. Basics

For a preliminary introduction to sharding, including how it works in Hibernate Search and what its limitations are, see Sharding and routing.

In the Lucene backend, sharding is disabled by default, but can be enabled by selecting a sharding strategy. Multiple strategies are available:

`hash`

```
# To configure the defaults for all indexes:
hibernate.search.backend.sharding.strategy = hash
hibernate.search.backend.sharding.number_of_shards = 2
# To configure a specific index:
hibernate.search.backend.indexes.<index-name>.sharding.strategy = hash
hibernate.search.backend.indexes.<index-name>.sharding.number_of_shards = 2
```

The `hash` strategy requires to set a number of shards through the `number_of_shards` property.

This strategy will set up an explicitly configured number of shards, numbered from 0 to the chosen number minus one (e.g. for 2 shards, there will be shard "0" and shard "1").

When routing, the routing key will be hashed to assign it to a shard. If the routing key is null, the document ID will be used instead.

This strategy is suitable when there is no explicit routing key configured in the mapping, or when the routing key has a large number of possible values that need to be brought down to a smaller number (e.g. "all integers").

`explicit`

```
# To configure the defaults for all indexes:
hibernate.search.backend.sharding.strategy = explicit
hibernate.search.backend.sharding.shard_identifiers = fr,en,de
# To configure a specific index:
hibernate.search.backend.indexes.<index-name>.sharding.strategy = explicit
hibernate.search.backend.indexes.<index-name>.sharding.shard_identifiers = fr,en,de
```

The `explicit` strategy requires to set a list of shard identifiers through the `shard_identifiers` property. The identifiers must be provided as a String containing multiple shard identifiers separated by commas, or a `Collection<String>` containing shard identifiers. A shard identifier can be any string.

This strategy will set up one shard per configured shard identifier.

When routing, the routing key will be validated to make sure it matches a shard identifier exactly. If it does, the document will be routed to that shard. If it does not, an exception will be thrown. The routing key cannot be null, and the document ID will be ignored.

This strategy is suitable when there is an explicit routing key configured in the mapping, and that routing key has a limited number of possible values that are known before starting the application.

17.3.2. Per-shard configuration

In some cases, in particular when using the `explicit` sharding strategy, it may be necessary to configure some shards in a slightly different way. For example, one of the shards may hold massive, but infrequently-accessed data, which should be stored on a different drive.

This can be achieved by adding configuration properties for a specific shard:

```
# Default configuration for all shards an index:
```

```
hibernate.search.backend.indexes.<index-name>.directory.root = /path/to/fast/drive/
# Configuration for a specific shard:
hibernate.search.backend.indexes.<index-name>.shards.<shard-identifier>.directory.root = /path/to/large/drive/
```

Not all settings can be overridden per shard; for example you can't override the sharding strategy on a per-shard basis.

Per-shard overriding is primarily intended for settings related to the directory and I/O.

Valid shard identifiers depend on the sharding strategy:

- For the `hash` strategy, each shard is assigned a positive integer, from `0` to the chosen number of shards minus one.
- For the `explicit` strategy, each shard is assigned one of the identifiers defined with the `shard_identifiers` property.

17.4. Index format compatibility

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

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

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

This configuration property is set at the backend level:

```
hibernate.search.backend.lucene_version = LUCENE_8_1_1
```

Depending on the specific version of Lucene you're using, you might have different options available: see `org.apache.lucene.util.Version` contained in `lucene-core.jar` for a list of allowed values.

When this option is not set, Hibernate Search will instruct Lucene to use the latest version, which is usually the best option for new projects. Still, it's recommended to define the version you're using explicitly in the configuration, so that when you happen to upgrade, Lucene the analyzers will not change behavior. You can then choose to update this value at a later time, for example when you have the chance to rebuild the index from scratch.

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

For information about which versions of Hibernate Search you can upgrade to, while retaining backward compatibility with a given version of Lucene APIs, refer to the compatibility policy.

17.5. Schema

Lucene does not really have a concept of centralized schema to specify the data type and capabilities of each field, but Hibernate Search maintains such a schema in memory, in order to remember which predicates/projections/sorts can be applied to each field.

For the most part, the schema is inferred from the mapping configured through Hibernate Search's mapping APIs, which are generic and independent of Lucene.

Aspects that are specific to the Lucene backend are explained in this section.

17.5.1. Field types

Available field types

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

Field types that are not in this list can still be used with a bit more work:

- If a property in the entity model has an unsupported type, but can be converted to a supported type, you will need a bridge. See Binding and bridges.
- If you need an index field with a specific type that is not supported by Hibernate Search, you will need a bridge that defines a native field type. See Index field type DSL extensions.

Table 17.1: Field types supported by the Lucene backend

Field type	Limitations
`java.lang.String`	-
`java.lang.Byte`	-
`java.lang.Short`	-
`java.lang.Integer`	-
`java.lang.Long`	-
`java.lang.Double`	-
`java.lang.Float`	-
`java.lang.Boolean`	-
`java.math.BigDecimal`	-
`java.math.BigInteger`	-
`java.time.Instant`	Lower range/resolution
`java.time.LocalDate`	Lower range/resolution
`java.time.LocalTime`	Lower range/resolution
`java.time.LocalDateTime`	Lower range/resolution
`java.time.ZonedDateTime`	Lower range/resolution
`java.time.OffsetDateTime`	Lower range/resolution
`java.time.OffsetTime`	Lower range/resolution
`java.time.Year`	Lower range/resolution
`java.time.YearMonth`	Lower range/resolution
`java.time.MonthDay`	-
`org.hibernate.search.<wbr>engine.<wbr>spatial.<wbr>GeoPoint`	Lower resolution

Range and resolution of date/time fields

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

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

Values that are out of range will trigger indexing failures.

Resolution for time types is also lower:

- `java.time` supports nanosecond-resolution.
- The Lucene backend supports millisecond-resolution.

Precision beyond the millisecond will be lost when indexing.

Range and resolution of `GeoPoint` fields

`GeoPoint`s are indexed as `LatLonPoint`s in the Lucene backend. According to `LatLonPoint`'s javadoc, there is a loss of precision when the values are encoded:

> *Values are indexed with some loss of precision from the original `double` values (`4.190951585769653E-8` for the latitude component and `8.381903171539307E-8` for longitude).*

This effectively means indexed points can be off by about 13 centimeters (5.2 inches) in the worst case.

Index field type DSL extensions

Not all Lucene field types have built-in support in Hibernate Search. Unsupported field types can still be used, however, by taking advantage of the "native" field type. Using this field type, Lucene `IndexableField` instances can be created directly, giving access to everything Lucene can offer.

Below is an example of how to use the Lucene "native" type.

Example 17.1: Using the Lucene "native" type

```
public class PageRankValueBinder implements ValueBinder { ①
    @Override
    public void bind(ValueBindingContext<?> context) {
        context.bridge(
                Float.class,
                new PageRankValueBridge(),
                context.typeFactory() ②
                        .extension( LuceneExtension.get() ) ③
                        .asNative( ④
                                Float.class, ⑤
                                (absoluteFieldPath, value, collector) -> { ⑥
                                    collector.accept( new FeatureField( absoluteFieldPath, "pageRank", value )
);
                                    collector.accept( new StoredField( absoluteFieldPath, value ) );
                                },
                                field -> (Float) field.numericValue() ⑦
                        )
        );
    }

    private static class PageRankValueBridge implements ValueBridge<Float, Float> {
        @Override
        public Float toIndexedValue(Float value, ValueBridgeToIndexedValueContext context) {
            return value; ⑧
        }

        @Override
        public Float fromIndexedValue(Float value, ValueBridgeFromIndexedValueContext context) {
            return value; ⑧
        }
    }
}
```

① Define a custom binder and its bridge. The "native" type can only be used from a binder, it cannot be used directly with annotation mapping. Here we're defining a value binder, but a type binder, or a property binder would work as well.

② Get the context's type factory.

③ Apply the Lucene extension to the type factory.

④ Call `asNative` to start defining a native type.

⑤ Define the field value type.

⑥ Define the `LuceneFieldContributor`. The contributor will be called upon indexing to add as many fields as necessary to the document. All fields must be named after the `absoluteFieldPath` passed to the contributor.

⑦ Optionally, if projections are necessary, define the `LuceneFieldValueExtractor`. The extractor will be called upon projecting to extract the projected value from a **single** stored field.

⑧ The value bridge is free to apply a preliminary conversion before passing the value to Hibernate Search, which will pass it along to the `LuceneFieldContributor`.

```
@Entity
@Indexed
public class WebPage {

    @Id
    private Integer id;

    @NonStandardField( ①
            valueBinder = @ValueBinderRef(type = PageRankValueBinder.class) ②
    )
    private Float pageRank;

    // Getters and setters
    // ...

}
```

① Map the property to an index field. Note that value bridges using a non-standard field type (such as Lucene's "native" type) must be mapped using the `@NonStandardField` annotation: other annotations such as `@GenericField` will fail.

② Instruct Hibernate Search to use our custom value binder.

17.5.2. Multi-tenancy

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

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

The multi-tenancy is automatically enabled in the backend if it is enabled in the mapper, e.g. if a multi-tenancy strategy is selected in Hibernate ORM, or if multi-tenancy is explicitly configured in the Standalone POJO mapper.

However, it is possible to enable multi-tenancy manually.

The multi-tenancy strategy is set at the backend level:

```
hibernate.search.backend.multi_tenancy.strategy = none
```

See the following subsections for details about available strategies.

`none`: single-tenancy

The `none` strategy (the default) disables multi-tenancy completely.

Attempting to set a tenant ID will lead to a failure when indexing.

`discriminator`: type name mapping using the index name

With the `discriminator` strategy, all documents from all tenants are stored in the same index.

When indexing, a discriminator field holding the tenant ID is populated transparently for each document.

When searching, a filter targeting the tenant ID field is added transparently to the search query to only return search hits for the current tenant.

17.6. Analysis

17.6.1. Basics

Analysis is the text processing performed by analyzers, both when indexing (document processing) and when searching (query processing).

The Lucene backend comes with some default analyzers, but analysis can also be configured explicitly.

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

1. Define a class that implements the `org.hibernate.search.backend.lucene.analysis.LuceneAnalysisConfigurer` interface.
2. Configure the backend to use that implementation by setting the configuration property `hibernate.search.backend.analysis.configurer` to a bean reference pointing to the implementation, for example `class:com.mycompany.MyAnalysisConfigurer`.

You can pass multiple bean references separated by commas. See Type of configuration properties.

Hibernate Search will call the `configure` method of this implementation on startup, and the configurer will be able to take advantage of a DSL to define analyzers and normalizers or even (for more advanced use) the similarity. See below for examples.

17.6.2. Built-in analyzers

Built-in analyzers are available out-of-the-box and don't require explicit configuration. If necessary, they can be overridden by defining your own analyzer with the same name.

The Lucene backend comes with a series of built-in analyzer; their names are listed as constants in `org.hibernate.search.engine.backend.analysis.AnalyzerNames`:

`default`

: The analyzer used by default with `@FullTextField`.

  Default implementation: `org.apache.lucene.analysis.standard.StandardAnalyzer`.

  Default behavior: first, tokenize using the standard tokenizer, which follows Word Break rules from the Unicode Text Segmentation algorithm, as specified in Unicode Standard Annex #29. Then, lowercase each token.

`standard`

: Default implementation: `org.apache.lucene.analysis.standard.StandardAnalyzer`.

  Default behavior: first, tokenize using the standard tokenizer, which follows Word Break rules from the Unicode Text Segmentation algorithm, as specified in Unicode Standard Annex #29. Then, lowercase each token.

`simple`

: Default implementation: `org.apache.lucene.analysis.core.SimpleAnalyzer`.

  Default behavior: first, split the text at non-letter characters. Then, lowercase each token.

`whitespace`

: Default implementation: `org.apache.lucene.analysis.core.WhitespaceAnalyzer`.

  Default behavior: split the text at whitespace characters. Do not change the tokens.

`stop`

: Default implementation: `org.apache.lucene.analysis.core.StopAnalyzer`.

  Default behavior: first, split the text at non-letter characters. Then, lowercase each token. Finally, remove English stop words.

`keyword`

: Default implementation: `org.apache.lucene.analysis.core.KeywordAnalyzer`.

  Default behavior: do not change the text in any way.

  With this analyzer a full text field would behave similarly to a keyword field, but with fewer features: no terms aggregations, for example.

  Consider using a `@KeywordField` instead.

17.6.3. Built-in normalizers

The Lucene backend does not provide any built-in normalizer.

17.6.4. Custom analyzers and normalizers

Referencing components by name

The context passed to the configurer exposes a DSL to define analyzers and normalizers:

Example 17.2: Implementing and using an analysis configurer to define analyzers and normalizers with the Lucene backend

```
package org.hibernate.search.documentation.analysis;

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

public class MyLuceneAnalysisConfigurer implements LuceneAnalysisConfigurer {
    @Override
    public void configure(LuceneAnalysisConfigurationContext context) {
        context.analyzer( "english" ).custom() ①
                .tokenizer( "standard" ) ②
                .charFilter( "htmlStrip" ) ③
                .tokenFilter( "lowercase" ) ④
                .tokenFilter( "snowballPorter" ) ④
                        .param( "language", "English" ) ⑤
                .tokenFilter( "asciiFolding" );

        context.normalizer( "lowercase" ).custom() ⑥
                .tokenFilter( "lowercase" )
                .tokenFilter( "asciiFolding" );

        context.analyzer( "french" ).custom() ⑦
                .tokenizer( "standard" )
                .charFilter( "htmlStrip" )
                .tokenFilter( "lowercase" )
                .tokenFilter( "snowballPorter" )
                        .param( "language", "French" )
                .tokenFilter( "asciiFolding" );
    }
}
```

① Define a custom analyzer named "english", because it will be used to analyze English text such as book titles.
② Set the tokenizer to a standard tokenizer: components are referenced by their name.
③ Set the char filters. Char filters are applied in the order they are given, before the tokenizer.
④ Set the token filters. Token filters are applied in the order they are given, after the tokenizer.
⑤ Set the value of a parameter for the last added char filter/tokenizer/token filter.
⑥ Normalizers are defined in a similar way, the only difference being that they cannot use a tokenizer.
⑦ Multiple analyzers/normalizers can be defined in the same configurer.

```
①
hibernate.search.backend.analysis.configurer =
class:org.hibernate.search.documentation.analysis.MyLuceneAnalysisConfigurer
```

① Assign the configurer to the backend using a Hibernate Search configuration property.

To know which character filters, tokenizers and token filters are available, you can call `context.availableTokenizers()`, `context.availableTokenizers()` and `context.availableTokenFilters()` on the context passed to your analysis configurer; this will return a set of all valid names.

To learn more about the behavior of these character filters, tokenizers and token filters, either browse the Lucene Javadoc, in particular, look through various packages of common analysis components, or read the corresponding section on the Solr Wiki (you don't need Solr to use these analyzers, it's just that there is no documentation page for Lucene proper).

In the Lucene Javadoc, the description of each factory class includes the mention "SPI Name" followed by a string constant. This is the name that should be passed to use that factory when defining analyzers.

Referencing components by factory class

Instead of names, you may also pass Lucene factory classes to refer to a particular tokenizer, char filter or token filter implementation. Those classes extend `org.apache.lucene.analysis.TokenizerFactory`, `org.apache.lucene.analysis.TokenFilterFactory` or `org.apache.lucene.analysis.CharFilterFactory`.

This avoids string constants in your code, at the cost of having a direct compile-time dependency to Lucene.

Example 17.3: Analysis configurer implementation using Lucene factory classes

```
context.analyzer( "english" ).custom()
        .tokenizer( StandardTokenizerFactory.class )
        .charFilter( HTMLStripCharFilterFactory.class )
        .tokenFilter( LowerCaseFilterFactory.class )
        .tokenFilter( SnowballPorterFilterFactory.class )
                .param( "language", "English" )
        .tokenFilter( ASCIIFoldingFilterFactory.class );

context.normalizer( "lowercase" ).custom()
        .tokenFilter( LowerCaseFilterFactory.class )
        .tokenFilter( ASCIIFoldingFilterFactory.class );
```

To know which character filters, tokenizers and token filters are available, either browse the Lucene Javadoc, in particular, look through various packages of common analysis components, or read the corresponding section on the Solr Wiki (you don't need Solr to use these analyzers, it's just that there is no documentation page for Lucene proper).

Assigning names to analyzer instances

It is also possible to assign a name to an analyzer instance:

Example 17.4: Naming an analyzer instance in the Lucene backend

```
context.analyzer( "my-standard" ).instance( new StandardAnalyzer() );
```

17.6.5. Overriding the default analyzer

The default analyzer when using `@FullTextField` without specifying an analyzer explicitly, is named `default`.

Like any other built-in analyzer, it is possible to override the default analyzer by defining a custom analyzer with the same name:

Example 17.5: Overriding the default analyzer in the Lucene backend

```
package org.hibernate.search.documentation.analysis;

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

public class DefaultOverridingLuceneAnalysisConfigurer implements LuceneAnalysisConfigurer {
    @Override
    public void configure(LuceneAnalysisConfigurationContext context) {
        context.analyzer( AnalyzerNames.DEFAULT ) ①
                .custom() ②
                .tokenizer( "standard" )
                .tokenFilter( "lowercase" )
                .tokenFilter( "snowballPorter" )
                        .param( "language", "French" )
                .tokenFilter( "asciiFolding" );
    }
}
```

① Start the definition of a custom analyzer that happens to be named `default`. Here we rely on constants from `org.hibernate.search.engine.backend.analysis.AnalyzerNames` to use the correct name, but hardcoding `"default"` would work just as well.

② Continue the analyzer definition as we would for any other custom analyzer.

```
①
hibernate.search.backend.analysis.configurer =
class:org.hibernate.search.documentation.analysis.DefaultOverridingLuceneAnalysisConfigurer
```

① Assign the configurer to the backend using a Hibernate Search configuration property.

17.6.6. Similarity

When searching, scores are assigned to documents based on statistics recorded at index time, using a specific formula. Those statistics and the formula are defined by a single component called the similarity, implementing Lucene's `org.apache.lucene.search.similarities.Similarity` interface.

By default, Hibernate Search uses `BM25Similarity` with its defaults parameters (`k1 = 1.2`, `b = 0.75`). This should provide satisfying scoring in most situations.

If you have advanced needs, you can set a custom `Similarity` in your analysis configurer, as shown below.

Remember to also reference the analysis configurer from your configuration properties, as explained in Custom analyzers and normalizers.

Example 17.6: Implementing an analysis configurer to change the Similarity with the Lucene backend

```
public class CustomSimilarityLuceneAnalysisConfigurer implements LuceneAnalysisConfigurer {
    @Override
    public void configure(LuceneAnalysisConfigurationContext context) {
        context.similarity( new ClassicSimilarity() ); ①

        context.analyzer( "english" ).custom() ②
                .tokenizer( "standard" )
                .tokenFilter( "lowercase" )
                .tokenFilter( "asciiFolding" );
    }
}
```

① Set the similarity to `ClassicSimilarity`.
② Define analyzers and normalizers as usual.

For more information about `Similarity`, its various implementations, and the pros and cons of each implementation, see the javadoc of `Similarity` and Lucene's source code.

You can also find useful resources on the web, for example in Elasticsearch's documentation.

17.7. Threads

The Lucene backend relies on an internal thread pool to execute write operations on the index.

By default, the pool contains exactly as many threads as the number of processors available to the JVM on bootstrap. That can be changed using a configuration property:

```
hibernate.search.backend.thread_pool.size = 4
```

This number is *per backend*, not per index. Adding more indexes will not add more threads.

Operations happening in this thread-pool include blocking I/O, so raising its size above the number of processor cores available to the JVM can make sense and may improve performance.

17.8. Indexing queues

Among all the write operations performed by Hibernate Search on the indexes, it is expected that there will be a lot of "indexing" operations to create/update/delete a specific document. We generally want to preserve the relative order of these requests when they are about the same documents.

For this reason, Hibernate Search pushes these operations to internal queues and applies them in batches. Each index maintains 10 queues holding at most 1000 elements each. Queues operate independently (in parallel), but each queue applies one operation after the other, so at any given time there can be at most 10 batches of indexing requests being applied for each index.

Indexing operations relative to the same document ID are always pushed to the same queue.

It is possible to customize the queues in order to reduce resource consumption, or on the contrary to improve throughput. This is done through the following configuration properties:

```
# To configure the defaults for all indexes:
hibernate.search.backend.indexing.queue_count = 10
hibernate.search.backend.indexing.queue_size = 1000
# To configure a specific index:
hibernate.search.backend.indexes.<index-name>.indexing.queue_count = 10
hibernate.search.backend.indexes.<index-name>.indexing.queue_size = 1000
```

- `indexing.queue_count` defines the number of queues. Expects a strictly positive integer value. The default for this property is `10`.

  Higher values will lead to more indexing operations being performed in parallel, which may lead to higher indexing throughput if CPU power is the bottleneck when indexing.

  Note that raising this number above the number of threads is never useful, as the number of threads limits how many queues can be processed in parallel.

- `indexing.queue_size` defines the maximum number of elements each queue can hold. Expects a strictly positive integer value. The default for this property is `1000`.

  Lower values may lead to lower memory usage, especially if there are many queues, but values that are too low will increase the likeliness of application threads blocking because the queue is full, which may lead to lower indexing throughput.

When a queue is full, any attempt to request indexing will block until the request can be put into the queue.

In order to achieve a reasonable level of performance, be sure to set the size of queues to a high enough number that this kind of blocking only happens when the application is under very high load.

When sharding is enabled, each shard will be assigned its own set of queues.

If you use the `hash` sharding strategy **based on the document ID** (and not based on a provided routing key), make sure to set the number of queues to a number with no common denominator with the number of shards; otherwise, some queues may be used much less than others.

For example, if you set the number of shards to 8 and the number of queues to 4, documents ending up in the shard #0 will always end up in queue #0 of that shard. That's because both the routing to a shard and the routing to a queue take the hash of the document ID then apply a modulo operation to that hash, and `<some hash> % 8 == 0` (routed to shard #0) implies `<some hash> % 4 == 0` (routed to queue #0 of shard #0). Again, this is only true if you rely on the document ID and not on a provided routing key for sharding.

17.9. Writing and reading

17.9.1. Commit

For a preliminary introduction to writing to and reading from indexes in Hibernate Search, including in particular the concepts of *commit* and *refresh*, see Commit and refresh.

In Lucene terminology, a *commit* is when changes buffered in an index writer are pushed to the index itself, so that a crash or power loss will no longer result in data loss.

Some operations are critical and are always committed before they are considered complete. This is the case for changes triggered by listener-triggered indexing (unless configured otherwise), and also for large-scale operations such as a purge. When such an operation is encountered, a commit will be performed immediately, guaranteeing that the operation is only considered complete after all changes are safely stored on disk.

However, other operations, like changes contributed by the mass indexer or when indexing is using the `async` synchronization strategy, are not expected to be committed immediately.

Performance-wise, committing may be an expensive operation, which is why Hibernate Search tries not to commit too often. By default, when changes that do not require an immediate commit are applied to the index, Hibernate Search will delay the commit by one second. If other changes are applied during that second, they will be included in the same commit. This dramatically reduces the amount of commits in write-intensive scenarios (e.g. mass indexing), leading to much better performance.

It is possible to control exactly how often Hibernate Search will commit by setting the commit interval (in milliseconds):

```
# To configure the defaults for all indexes:
```

```
hibernate.search.backend.io.commit_interval = 1000
# To configure a specific index:
hibernate.search.backend.indexes.<index-name>.io.commit_interval = 1000
```

The default for this property is `1000`.

Setting the commit interval to 0 will force Hibernate Search to commit after every batch of changes, which may result in a much lower throughput, in particular for explicit or listener-triggered indexing but even more so for mass indexing.

Remember that individual write operations may force a commit, which may cancel out the potential performance gains from setting a higher commit interval.

By default, the commit interval may only improve throughput of the mass indexer. If you want changes triggered explicitly or by listeners to benefit from it too, you will need to select a non-default synchronization strategy, so as not to require a commit after each change.

17.9.2. Refresh

For a preliminary introduction to writing to and reading from indexes in Hibernate Search, including in particular the concepts of *commit* and *refresh*, see Commit and refresh.

In Lucene terminology, a *refresh* is when a new index reader is opened, so that the next search queries will take into account the latest changes to the index.

Performance-wise, refreshing may be an expensive operation, which is why Hibernate Search tries not to refresh too often. The index reader is refreshed upon every search query, but only if writes have occurred since the last refresh.

In write-intensive scenarios where refreshing after each write is still too frequent, it is possible to refresh less frequently and thus improve read throughput by setting a refresh interval in milliseconds. When set to a value higher than 0, the index reader will no longer be refreshed upon every search query: if, when a search query starts, the refresh occurred less than X milliseconds ago, then the index reader will not be refreshed, even though it may be out-of-date.

The refresh interval can be set this way:

```
# To configure the defaults for all indexes:
hibernate.search.backend.io.refresh_interval = 0
# To configure a specific index:
hibernate.search.backend.indexes.<index-name>.io.refresh_interval = 0
```

The default for this property is `0`.

17.9.3. `IndexWriter` settings

Lucene's `IndexWriter`, used by Hibernate Search to write to indexes, exposes several settings that can be tweaked to better fit your application, and ultimately get better performance.

Hibernate Search exposes these settings through configuration properties prefixed with `io.writer.`, at the index level.

Below is a list of all index writer settings. They can all be set similarly through configuration properties; for example, `io.writer.ram_buffer_size` can be set like this:

```
# To configure the defaults for all indexes:
hibernate.search.backend.io.writer.ram_buffer_size = 32
# To configure a specific index:
hibernate.search.backend.indexes.<index-name>.io.writer.ram_buffer_size = 32
```

Table 17.2: Configuration properties for the `IndexWriter`

Property	Description
`[…].io.writer.max_buffered_docs`	The maximum number of documents that can be buffered in-memory before they are flushed to the Directory. Large values mean faster indexing, but more RAM usage. When used together with `ram_buffer_size` a flush occurs for whichever event happens first.
`[…].io.writer.ram_buffer_size`	The maximum amount of RAM that may be used for buffering added documents and deletions before they are flushed to the Directory. Large values mean faster indexing, but more RAM usage. Generally for faster indexing performance it's best to use this setting rather than `max_buffered_docs`. When used together with `max_buffered_docs` a flush occurs for whichever event happens first.
`[…].io.writer.infostream`	Enables low level trace information about Lucene's internal components; `true` or `false`. Logs will be appended to the logger `org.hibernate.search.lucene.infostream` at the `TRACE` level. This may cause significant performance degradation, even if the logger ignores the TRACE level, so this should only be used for troubleshooting purposes. Disabled by default.

Refer to Lucene's documentation, in particular the javadoc and source code of `IndexWriterConfig`, for more information about the settings and their defaults.

17.9.4. Merge settings

A Lucene index is not stored in a single, continuous file. Instead, each flush to the index will generate a small file containing all the documents added to the index. This file is called a "segment". Search can be slower on an index with too many segments, so Lucene regularly merges small segments to create fewer, larger segments.

Lucene's merge behavior is controlled through a `MergePolicy`. Hibernate Search uses the `LogByteSizeMergePolicy`, which exposes several settings that can be tweaked to better fit your application, and ultimately get better performance.

Below is a list of all merge settings. They can all be set similarly through configuration properties; for example, `io.merge.factor` can be set like this:

```
# To configure the defaults for all indexes:
hibernate.search.backend.io.merge.factor = 10
# To configure a specific index:
hibernate.search.backend.indexes.<index-name>.io.merge.factor = 10
```

Table 17.3: Configuration properties related to merges

Property	Description
`[…].io.merge.max_docs`	The maximum number of documents that a segment can have before merging. Segments with more than this number of documents will not be merged. Smaller values perform better on frequently changing indexes, larger values provide better search performance if the index does not change often.
`[…].io.merge.factor`	The number of segments that are merged at once. With smaller values, merging happens more often and thus uses more resources, but the total number of segments will be lower on average, increasing read performance. Thus, larger values (`> 10`) are best for mass indexing, and smaller values (`< 10`) are best for explicit or listener-triggered indexing. The value must not be lower than `2`.

Property	Description
`[…].io.merge.min_size`	The minimum target size of segments, in MB, for background merges. Segments smaller than this size are merged more aggressively. Setting this too large might result in expensive merge operations, even tough they are less frequent.
`[…].io.merge.max_size`	The maximum size of segments, in MB, for background merges. Segments larger than this size are never merged in the background. Settings this to a lower value helps reduce memory requirements and avoids some merging operations at the cost of optimal search speed. When forcefully merging an index, this value is ignored and `max_forced_size` is used instead (see below).
`[…].io.merge.max_forced_size`	The maximum size of segments, in MB, for forced merges. This is the equivalent of `io.merge.max_size` for forceful merges. You will generally want to set this to the same value as `max_size` or lower, but setting it too low will degrade search performance as documents are deleted.
`[…].io.merge.calibrate_by_deletes`	Whether the number of deleted documents in an index should be taken into account; `true` or `false`. When enabled, Lucene will consider that a segment with 100 documents, 50 of which are deleted, actually contains 50 documents. When disabled, Lucene will consider that such a segment contains 100 documents. Setting `calibrate_by_deletes` to `false` will lead to more frequent merges caused by `io.merge.max_docs`, but will more aggressively merge segments with many deleted documents, improving search performance.

Refer to Lucene's documentation, in particular the javadoc and source code of `LogByteSizeMergePolicy`, for more information about the settings and their defaults.

The options `io.merge.max_size` and `io.merge.max_forced_size` do not **directly** define the maximum size of all segment files.

First, consider that merging a segment is about adding it together with another existing segment to form a larger one. `io.merge.max_size` is the maximum size of segments **before** merging, so newly merged segments can be up to twice that size.

Second, merge options do not affect the size of segments initially created by the index writer, before they are merged. This size can be limited with the setting `io.writer.ram_buffer_size`, but Lucene relies on estimates to implement this limit; when these estimates are off, it is possible for newly created segments to be slightly larger than `io.writer.ram_buffer_size`.

So, for example, to be fairly confident no file grows larger than 15MB, use something like this:

```
hibernate.search.backend.io.writer.ram_buffer_size = 10
hibernate.search.backend.io.merge.max_size = 7
hibernate.search.backend.io.merge.max_forced_size = 7
```

17.10. Searching

Searching with the Lucene backend relies on the same APIs as any other backend.

This section details Lucene-specific configuration related to searching.

17.10.1. Low-level hit caching

This feature implies that application code rely on Lucene APIs directly.

> An upgrade of Hibernate Search, even for a bugfix (micro) release, may require an upgrade of Lucene, which may lead to breaking API changes in Lucene.
>
> If this happens, you will need to change application code to deal with the changes.

Lucene supports caching low-level hits, i.e. caching the list of documents that match a given `org.apache.lucene.search.Query` in a given index segment.

This cache can be useful in read-intensive scenarios, where the same query is executed very often on the same index, and the index is rarely written to.

> This is different from Hibernate ORM caching, which caches the content of entities or the results of **database** queries. Hibernate ORM caching can also be leveraged in Hibernate Search, but through a different API: see Cache lookup strategy.

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

1. Define a class that implements the `org.hibernate.search.backend.lucene.cache.QueryCachingConfigurer` interface.
2. Configure the backend to use that implementation by setting the configuration property `hibernate.search.backend.query.caching.configurer` to a bean reference pointing to the implementation, for example `class:com.mycompany.MyQueryCachingConfigurer`.

> You can pass multiple bean references separated by commas. See Type of configuration properties.

Hibernate Search will call the `configure` method of this implementation on startup, and the configurer will be able to take advantage of a DSL to define the `org.apache.lucene.search.QueryCache` and the `org.apache.lucene.search.QueryCachingPolicy`.

Chapter 18. Elasticsearch backend

18.1. Compatibility

18.1.1. Overview

Hibernate Search's Elasticsearch backend is compatible with multiple distributions of Elasticsearch:

- Elasticsearch clusters running version 7.10+, 8.x or 9.x,
- OpenSearch clusters running version 1.3, 2.x or 3.x
- Amazon OpenSearch Service clusters running version 1.3, 2.x or 3.x (requires extra configuration)
- Amazon OpenSearch Serverless clusters (incubating, requires extra configuration)

For information about which versions of Hibernate Search are compatible with a given version of Elasticsearch/OpenSearch, refer to the compatibility matrix.

For information about which future versions of Hibernate Search you can expect to retain compatibility with currently compatible versions of Elasticsearch/OpenSearch, refer to the compatibility policy.

Where possible, the distribution and version running on your cluster will be automatically detected on startup, and Hibernate Search will adapt based on that.

With Amazon OpenSearch Serverless, or when your cluster is not available on startup, you will have to configure the version Hibernate Search should expect explicitly: see Version compatibility for details.

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

18.1.2. Elasticsearch

Hibernate Search's Elasticsearch backend is compatible with Elasticsearch clusters running version 7.10+, 8.x or 9.x and regularly tested against versions 7.10, 7.17, 8.18 or 9.4.

Using Elasticsearch currently doesn't require specific configuration and doesn't imply specific limitations.

18.1.3. OpenSearch

Hibernate Search's Elasticsearch backend is compatible with OpenSearch clusters running version 1.3, 2.x or 3.x and regularly tested against versions 1.3, 2.19 or 3.6.

Using OpenSearch currently doesn't require specific configuration There are some limitations applied by Hibernate Search when it comes to using a knn predicate with OpenSearch. These limitations come from OpenSearch's feature availability, see this section of the documentation for more details.

18.1.4. Amazon OpenSearch Service

Hibernate Search's Elasticsearch backend is compatible with Amazon OpenSearch Service and regularly tested against key versions.

Using Amazon OpenSearch Service requires proprietary authentication that involves extra configuration.

Using Amazon OpenSearch Service implies a single limitations: when running Elasticsearch (not OpenSearch) and only in version 7.1 or older, closing indexes is not possible, and as a result automatic schema updates (not recommended in production) will fail when trying to update analyzer definitions.

18.1.5. Amazon OpenSearch Serverless (incubating)

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

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

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

Amazon OpenSearch Serverless compatibility is implemented and incubating; feel free to provide feedback on HSEARCH-4867.

However, be aware that:

- Hibernate Search does not currently get tested against Amazon OpenSearch Serverless; see HSEARCH-4919.
- Connecting to an Amazon OpenSearch Serverless cluster requires proprietary authentication that involves extra configuration.
- Compatibility with Amazon OpenSearch Serverless must be enabled explicitly by setting `hibernate.search.backend.version` to `amazon-opensearch-serverless`.

Also, Amazon OpenSearch Serverless has its own, specific limitations:

- Closing indexes is not possible, and as a result automatic schema updates (not recommended in production) will fail when trying to update analyzer definitions.
- Distribution/version detection on startup is not possible, so it is disabled by default and cannot be enabled explicitly.
- Minimal index status requirement for schema management is not possible, so it is disabled by default and cannot be enabled explicitly.
- Purging, flushing, refreshing, or merging segments is not possible, so attempts to perform these operations explicitly will always fail.
- The mass indexer will fail if you attempt a purge on start (the default), because Amazon OpenSearch Serverless doesn't support it. Use `.dropAndCreateSchemaOnStart(…)` to drop the indexes on start instead. See HSEARCH-4930.
- The mass indexer will skip the flush, refresh and merge-segments operations by default, and attempting to enable them explicitly will result in failures, because Amazon OpenSearch Serverless doesn't support them.
- The Jakarta Batch integration is not currently supported. See HSEARCH-4929, HSEARCH-4930.

18.1.6. Upgrading Elasticsearch

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

On top of that, there might be some fundamental differences between some versions of Elasticsearch. Please refer to the Elasticsearch documentation and migration guides to identify any incompatible schema changes.

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

18.2. Basic configuration

All configuration properties of the Elasticsearch backend are optional, but the defaults might not suit everyone. In particular your production Elasticsearch cluster is probably not reachable at `http://localhost:9200`, so you will need to set the address of your cluster by configuring the client.

Configuration properties are mentioned in the relevant parts of this documentation. You can find a full reference of available properties in the Elasticsearch backend configuration properties appendix.

18.3. Configuration of the Elasticsearch cluster

Most of the time, Hibernate Search does not require any specific configuration to be applied by hand to the Elasticsearch cluster, beyond the index mapping (schema) which can be automatically generated.

The only exception is Sharding, which needs to be enabled explicitly.

18.4. Client configuration

An Elasticsearch backend communicates with an Elasticsearch cluster through a REST client. Hibernate Search comes with a range of pluggable Elasticsearch REST clients that you can pick from.

Currently, the default one is the `hibernate-search-backend-elasticsearch-client-rest4`, which may change in a future version of Hibernate Search. You should explicitly define the REST client in the application dependencies, if it matters which one the Elasticsearch backend will pick.

To change the REST client you need to add the dependency that contains the client and specify the client factory configuration property.

18.4.1. Available clients

Elasticsearch low level REST client (Apache HTTP Client 4)

Coordinates

GroupID

`org.hibernate.search`

ArtifactID

`hibernate-search-backend-elasticsearch-client-rest4`

Underlying HTTP Client

Apache HTTP Client 4

This Elasticsearch backend REST client is based on the Elasticsearch low level client (`org.elasticsearch.client:elasticsearch-rest-client`). Currently, the default REST client used by the Elasticsearch backend.

Elasticsearch low level REST client (Apache HTTP Client 5)

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

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

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

Coordinates

GroupID

`org.hibernate.search`

ArtifactID

`hibernate-search-backend-elasticsearch-client-rest5`

Underlying HTTP Client

Apache HTTP Client 5

This Elasticsearch backend REST client is based on the newer Elasticsearch low level client (`co.elastic.clients:elasticsearch-rest5-client`), which is also used in the `co.elastic.clients:elasticsearch-java`.

OpenSearch low level REST client (Apache HTTP Client 5)

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

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

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

Coordinates

GroupID

`org.hibernate.search`

ArtifactID

`hibernate-search-backend-elasticsearch-client-opensearch-rest`

Underlying HTTP Client

Apache HTTP Client 5

This Elasticsearch backend REST client is based on the OpenSearch low level client (`org.opensearch.client:opensearch-rest-client`).

Simple REST client based on the JDKs `HttpClient`

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

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

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

Coordinates

GroupID

`org.hibernate.search`

ArtifactID

`hibernate-search-backend-elasticsearch`

Underlying HTTP Client

JDKs Http Client (`java.net.http.HttpClient`)

This Elasticsearch backend REST client is based on the HTTP client that is part of the JDKs `java.net` API. Mostly intended for very basic use cases, and may lack some features, e.g. automatic node discovery, compared to the other clients. As this client implementation relies only on the JDK, it is shipped as part of the Elasticsearch backend and does not require any extra dependencies.

> JDKs HTTP client allows configuration though the system properties. Consult the list of available properties for your JDK version, e.g. JDK 25

18.4.2. Target hosts

The following property configures the Elasticsearch host (or hosts) to send indexing requests and search queries to:

```
hibernate.search.backend.hosts = localhost:9200
```

The default for this property is `localhost:9200`.

This property may be set to a String representing a host and port such as `localhost` or `es.mycompany.com:4400`, or a String containing multiple such host-and-port strings separated by commas, or a `Collection<String>` containing such host-and-port strings.

You may change the protocol used to communicate with the hosts using this configuration property:

```
hibernate.search.backend.protocol = http
```

The default for this property is `http`.

This property may be set to either `http` or `https`.

Alternatively, it is possible to define both the protocol and hosts as one or more URIs using a single property:

```
hibernate.search.backend.uris = http://localhost:9200
```

This property may be set to a String representing a URI such as `http://localhost` or `https://es.mycompany.com:4400`, or a String containing multiple such URI strings separated by commas, or a `Collection<String>` containing such URI strings.

> There are some constraints regarding the use of this property:
>
> - All the uris must have the same protocol.
> - Cannot be used if `hosts` or `protocol` are set.
> - The provided list of URIs must not be empty.

> Configuration properties from this section are applicable to the following clients:

Client	Configuration property applicability
Elasticsearch low level REST client	✔
Elasticsearch low level REST 5 client	✔
OpenSearch low level client	✔
JDK HttpClient based REST client	✔

18.4.3. Path prefix

By default, the REST API is expected to be available to the root path (`/`). For example a search query target all indexes will be sent to path `/_search`. This is what you need for a standard Elasticsearch setup.

If your setup is non-standard, for example because of a non-transparent proxy between the application and the Elasticsearch cluster, you can use a configuration similar to this:

```
hibernate.search.backend.path_prefix = my/path
```

With the above, a search query targeting all indexes will be sent to path `/my/path/_search` instead of `/_search`. The path will be prefixed similarly for all requests sent to Elasticsearch.

Configuration properties from this section are applicable to the following clients:

Client	Configuration property applicability
Elasticsearch low level REST client	✔
Elasticsearch low level REST 5 client	✔
OpenSearch low level client	✔
JDK HttpClient based REST client	✔

18.4.4. Node discovery

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

Automatic discovery is controlled by the following properties:

```
hibernate.search.backend.discovery.enabled = false
hibernate.search.backend.discovery.refresh_interval = 10
```

- `discovery.enabled` defines whether the feature is enabled. Expects a boolean value. The default for this property is `false`.
- `discovery.refresh_interval` defines the interval between two executions of the automatic discovery. Expects a positive integer, in seconds. The default for this property is `10`.

Configuration properties from this section are applicable to the following clients:

Client	Configuration property applicability
Elasticsearch low level REST client	✔
Elasticsearch low level REST 5 client	✔
OpenSearch low level client	✔
JDK HttpClient based REST client	✘

18.4.5. HTTP authentication

HTTP authentication is disabled by default, but may be enabled by setting the following configuration properties:

```
hibernate.search.backend.username = ironman
hibernate.search.backend.password = j@rv1s
```

The default for these properties is an empty string.

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

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

Configuration properties from this section are applicable to the following clients:

Client	Configuration property applicability
Elasticsearch low level REST client	✔
Elasticsearch low level REST 5 client	✔
OpenSearch low level client	✔
JDK HttpClient based REST client	✔

18.4.6. Authentication on Amazon Web Services

The Hibernate Search Elasticsearch backend, once configured, will work just fine in most setups. However, if you need to use Amazon OpenSearch Service or Amazon OpenSearch Serverless, you will find they require a proprietary authentication method: request signing.

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

You will need to add this dependency:

```
<dependency>
   <groupId>org.hibernate.search</groupId>
   <artifactId>hibernate-search-backend-elasticsearch-aws</artifactId>
   <version>8.4.0.CR1</version>
</dependency>
```

With that dependency in your classpath, you will still need to configure it.

The following configuration is mandatory:

```
hibernate.search.backend.aws.signing.enabled = true
hibernate.search.backend.aws.region = us-east-1
```

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

By default, Hibernate Search will rely on the default credentials provider from the AWS SDK. This provider will look for credentials in various places (Java system properties, environment variables, AWS-specific configuration, …). For more information about how the default credentials provider works, see its official documentation.

Optionally, you can set static credentials with the following options:

```
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
```

- `aws.credentials.type` defines the type of credentials. Set to `default` to get the default behavior (as explained above), or to `static` to provide credentials using the properties below.
- `aws.credentials.access_key_id` defines the access key ID. Expects a string value. This property has no default and must be provided when the credentials type is set to `static`.
- `aws.credentials.secret_access_key` defines the secret access key. Expects a string value. This property has no default and must be provided when the credentials type is set to `static`.

Configuration properties from this section are applicable to the following clients:

Client	Configuration property applicability
Elasticsearch low level REST client	✔

Client	Configuration property applicability
Elasticsearch low level REST 5 client	✔
OpenSearch low level client	✔
JDK HttpClient based REST client	✔

18.4.7. Connection tuning

Timeouts

```
# hibernate.search.backend.request_timeout = 30000
hibernate.search.backend.connection_timeout = 1000
hibernate.search.backend.read_timeout = 30000
```

- `request_timeout` defines the timeout when executing a request. This includes the time needed to establish a connection, send the request and read the response. This property is not defined by default.
- `connection_timeout` defines the timeout when establishing a connection. The default for this property is `1000`.
- `read_timeout` defines the timeout when reading a response. The default for this property is `30000`.

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

Configuration properties from this section are applicable to the following clients:

Client	Configuration property applicability
Elasticsearch low level REST client	✔
Elasticsearch low level REST 5 client	✔
OpenSearch low level client	✔
JDK HttpClient based REST client	✔ (Read timeout is not supported)

Connection pool

```
hibernate.search.backend.max_connections = 20
hibernate.search.backend.max_connections_per_route = 10
```

- `max_connections` defines maximum number of simultaneous connections to the Elasticsearch cluster, all hosts taken together. The default for this property is `40`.
- `max_connections_per_route` defines maximum number of simultaneous connections to each host of the Elasticsearch cluster. The default for this property is `10`.

These properties expect a positive Integer value, such as `20`.

When working with an Elasticsearch backend cluster, multiple cluster nodes might exist. If so, the REST client communicating with the cluster will try to distribute the requests among the nodes. That's why the maximum number of connections can be set per route (cluster node) and then limited by the general number of maximum connections.

When modifying the maximum connection values, it is worth remembering the number of indexing queues configured for the backend With an application performing a lot of indexing operations, having a subpar configuration of the maximum connections may result in all connections primarily being consumed by the indexing processes, resulting in the degrading performance of search operations, as those would be struggling to get a connection. To prevent that from happening consider having the maximum number of connections greater than the number of active indexing queues.

Configuration properties from this section are applicable to the following clients:

Client	Configuration property applicability
Elasticsearch low level REST client	✔
Elasticsearch low level REST 5 client	✔
OpenSearch low level client	✔
JDK HttpClient based REST client	✘ (Configurable through JDKs Http client system properties)

Keep Alive

```
hibernate.search.backend.max_keep_alive = 10000
```

- `max_keep_alive` defines how long connections to the Elasticsearch cluster can be kept idle.

  Expects a positive Long value in milliseconds, such as `60000`.

  If the response from an Elasticsearch cluster contains a `Keep-Alive` header, then the effective max idle time will be whichever is lower: the duration from the `Keep-Alive` header or the value of this property (if set).

  If this property is not set, only the `Keep-Alive` header is considered, and if it's absent, idle connections will be kept forever.

Configuration properties from this section are applicable to the following clients:

Client	Configuration property applicability
Elasticsearch low level REST client	✔
Elasticsearch low level REST 5 client	✔
OpenSearch low level client	✔
JDK HttpClient based REST client	✘ "(Configurable through JDKs Http client system properties)"

18.4.8. Custom HTTP client configurations

Each of the pluggable REST clients allow for additional configuration of the underlying HTTP client through configurers.

See the following sections to learn how each of the clients can be configured.

18.4.9. Elasticsearch low level client

It is possible to directly configure the underlying Apache HTTP 4 client of the Elasticsearch low level client using an instance of `org.apache.http.impl.nio.client.HttpAsyncClientBuilder`.

With this API you can add interceptors, change the keep alive, the max connections, the SSL key/trust store settings and many other client configurations.

Configuring the HTTP client directly requires two steps:

1. Define a class that implements the `org.hibernate.search.backend.elasticsearch.client.rest4.ElasticsearchHttpClientConfigurer` interface.
2. Configure Hibernate Search to use that implementation by setting the configuration property `hibernate.search.backend.client.configurer` to a bean reference pointing to the implementation, for example `class:org.hibernate.search.documentation.backend.elasticsearch.client.rest.HttpClientConfigurer`.

Example 18.1: Implementing and using a `ElasticsearchHttpClientConfigurer`

```
public class HttpClientConfigurer implements ElasticsearchHttpClientConfigurer { ①

    @Override
```

```
        public void configure(ElasticsearchHttpClientConfigurationContext context) { ②
            HttpAsyncClientBuilder clientBuilder = context.clientBuilder(); ③
            clientBuilder.setMaxConnPerRoute( 7 ); ④
            clientBuilder.addInterceptorFirst( (HttpResponseInterceptor) (response, httpContext) -> {
                long contentLength = response.getEntity().getContentLength();
                // doing some stuff with contentLength
            } );
        }
    }
```

① The class has to implement the `ElasticsearchHttpClientConfigurer` interface.
② The `configure` method provides the access to the `ElasticsearchHttpClientConfigurationContext`.
③ From the context it is possible to get the `HttpAsyncClientBuilder`.
④ Finally, you can use the builder to configure the client with your custom settings.

Example 18.2: Define a custom http client configurer in the properties

```
    ①
    hibernate.search.backend.client.configurer =
    class:org.hibernate.search.documentation.backend.elasticsearch.client.rest4.HttpClientConfigurer
```

① Specify the HTTP client configurer.

Any setting defined by a custom http client configurer will override any other setting defined by Hibernate Search.

18.4.10. Elasticsearch java client

It is possible to directly configure the underlying Apache HTTP 5 client of the Elasticsearch java client using an instance of `org.apache.hc.client5.http.impl.async.HttpAsyncClientBuilder`.

With this API you can add interceptors, change the keep alive, the max connections, the SSL key/trust store settings and many other client configurations.

Configuring the HTTP client directly requires two steps:

1. Define a class that implements the `org.hibernate.search.backend.elasticsearch.client.rest5.ElasticsearchHttpClientConfigurer` interface.
2. Configure Hibernate Search to use that implementation by setting the configuration property `hibernate.search.backend.client.configurer` to a bean reference pointing to the implementation, for example `class:org.hibernate.search.documentation.backend.elasticsearch.client.java.HttpClientConfigurer`.

Example 18.3: Implementing and using a `ElasticsearchHttpClientConfigurer`

```
    public class HttpClientConfigurer implements ElasticsearchHttpClientConfigurer { ①

        @Override
        public void configure(ElasticsearchHttpClientConfigurationContext context) { ②
            HttpAsyncClientBuilder clientBuilder = context.clientBuilder(); ③
            clientBuilder.setConnectionManager( ④
                    PoolingAsyncClientConnectionManagerBuilder.create()
                            .setMaxConnTotal( 20 )
                            .setMaxConnPerRoute( 10 )
                            .build()
            );
            clientBuilder.addResponseInterceptorFirst( (response, entityDetails, httpContext) -> {
                long contentLength = entityDetails.getContentLength();
                // doing some stuff with contentLength
            } );
        }
    }
```

① The class has to implement the `ElasticsearchHttpClientConfigurer` interface.
② The `configure` method provides the access to the `ElasticsearchHttpClientConfigurationContext`.
③ From the context it is possible to get the `HttpAsyncClientBuilder`.

④ Finally, you can use the builder to configure the client with your custom settings.

Example 18.4: Define a custom http client configurer in the properties

```
①
hibernate.search.backend.client.configurer =
class:org.hibernate.search.documentation.backend.elasticsearch.client.rest5.HttpClientConfigurer
```

① Specify the HTTP client configurer.

Any setting defined by a custom http client configurer will override any other setting defined by Hibernate Search.

18.4.11. OpenSearch low level client

It is possible to directly configure the underlying Apache HTTP 5 client of the OpenSearch low level client using an instance of `org.apache.hc.client5.http.impl.async.HttpAsyncClientBuilder`.

With this API you can add interceptors, change the keep alive, the max connections, the SSL key/trust store settings and many other client configurations.

Configuring the HTTP client directly requires two steps:

1. Define a class that implements the `org.hibernate.search.backend.elasticsearch.client.opensearch.rest.ElasticsearchHttpClientConfigurer` interface.
2. Configure Hibernate Search to use that implementation by setting the configuration property `hibernate.search.backend.client.configurer` to a bean reference pointing to the implementation, for example `class:org.hibernate.search.documentation.backend.elasticsearch.client.opensearch.HttpClientConfigurer`.

Example 18.5: Implementing and using a `ElasticsearchHttpClientConfigurer`

```
public class HttpClientConfigurer implements ElasticsearchHttpClientConfigurer { ①

    @Override
    public void configure(ElasticsearchHttpClientConfigurationContext context) { ②
        HttpAsyncClientBuilder clientBuilder = context.clientBuilder(); ③
        clientBuilder.setConnectionManager( ④
                PoolingAsyncClientConnectionManagerBuilder.create()
                        .setMaxConnTotal( 20 )
                        .setMaxConnPerRoute( 10 )
                        .build()
        );
        clientBuilder.addResponseInterceptorFirst( (response, entityDetails, httpContext) -> {
            long contentLength = entityDetails.getContentLength();
            // doing some stuff with contentLength
        } );
    }
}
```

① The class has to implement the `ElasticsearchHttpClientConfigurer` interface.
② The `configure` method provides the access to the `ElasticsearchHttpClientConfigurationContext`.
③ From the context it is possible to get the `HttpAsyncClientBuilder`.
④ Finally, you can use the builder to configure the client with your custom settings.

Example 18.6: Define a custom http client configurer in the properties

```
①
hibernate.search.backend.client.configurer =
class:org.hibernate.search.documentation.backend.elasticsearch.client.opensearch.HttpClientConfigurer
```

① Specify the HTTP client configurer.

Any setting defined by a custom http client configurer will override any other setting defined by Hibernate Search.

18.4.12. JDK `HttpClient` based REST client

It is possible to directly configure the underlying `HttpClient` using an instance of `java.net.http.HttpClient.Builder`.

Configuring the HTTP client directly requires two steps:

1. Define a class that implements the `org.hibernate.search.backend.elasticsearch.client.jdk.ElasticsearchHttpClientConfigurer` interface.
2. Configure Hibernate Search to use that implementation by setting the configuration property `hibernate.search.backend.client.configurer` to a bean reference pointing to the implementation, for example `class:org.hibernate.search.documentation.backend.elasticsearch.client.jdk.HttpClientConfigurer`.

Example 18.7: Implementing and using a `ElasticsearchHttpClientConfigurer`

```
public class HttpClientConfigurer implements ElasticsearchHttpClientConfigurer { ①

    @Override
    public void configure(ElasticsearchHttpClientConfigurationContext context) { ②
        HttpClient.Builder builder = context.clientBuilder(); ③
        builder.connectTimeout( Duration.ofSeconds( 2 ) ); ④
    }
}
```

① The class has to implement the `ElasticsearchHttpClientConfigurer` interface.
② The `configure` method provides the access to the `ElasticsearchHttpClientConfigurationContext`.
③ From the context it is possible to get the `HttpClient.Builder`.
④ Finally, you can use the builder to configure the client with your custom settings.

Example 18.8: Define a custom http client configurer in the properties

```
①
hibernate.search.backend.client.configurer =
class:org.hibernate.search.documentation.backend.elasticsearch.client.jdk.HttpClientConfigurer
```

① Specify the HTTP client configurer.

Any setting defined by a custom http client configurer will override any other setting defined by Hibernate Search.

18.5. Version compatibility

18.5.1. Version assumed by Hibernate Search

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

By default, Hibernate Search will query the Elasticsearch/OpenSearch cluster at boot time to retrieve this information, and will infer the correct behavior to adopt.

You can force Hibernate Search to expect a specific version of Elasticsearch/OpenSearch by setting the property `hibernate.search.backend.version` to a version string following the format `x.y.z-qualifier` or `<distribution>:x.y.z-qualifier` or just `<distribution>`, where:

- `<distribution>` is either `elastic`, `opensearch` or `amazon-opensearch-serverless`. Optional, defaults to `elastic`.
- `x`, `y` and `z` are integers. `x` is mandatory, `y` and `z` are optional.
- `qualifier` is a string of word characters (alphanumeric or _). Optional.

For example, `8`, `8.0`, `8.9`, `opensearch:2.9`, `amazon-opensearch-service` are all valid version strings.

Amazon OpenSearch Serverless is a special case as it doesn't use version numbers.

> When using that platform, you **must** set the version and it must be set to simply `amazon-opensearch-serverless`, without a trailing `:` or version number.

Hibernate Search will still query the Elasticsearch/OpenSearch cluster to detect the actual distribution and version of the cluster (except where not supported, i.e. Amazon OpenSearch Serverless), in order to check that the configured distribution and version match the actual ones.

18.5.2. Disabling the version check on startup

If necessary, you can disable the call to the Elasticsearch/OpenSearch cluster on startup, and provide the information manually.

To do that, set the property `hibernate.search.backend.version_check.enabled` to `false`.

You will also have to set the property `hibernate.search.backend.version` to a version string as explained in the previous section.

In this case, both major **and minor** version numbers (`x` **and** `y` in the formats above) are mandatory, but the `distribution` can be left out if it is the default (`elasticsearch`), and all other components (micro, qualifier) remain optional. For example, `8.0`, `8.9`, `opensearch:2.9` are all valid version strings in this case, but `8` is not precise enough.

18.6. Request logging

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

18.7. Sharding

> For a preliminary introduction to sharding, including how it works in Hibernate Search and what its limitations are, see Sharding and routing.

Elasticsearch disables sharding by default. To enable it, set the property `index.number_of_shards` in your cluster.

18.8. Schema management

Elasticsearch indexes need to be created before they can be used for indexing and searching; see Managing the index schema for more information about how to create indexes and their schema in Hibernate Search.

For Elasticsearch specifically, some fine-tuning is available through the following options:

```
# To configure the defaults for all indexes:
hibernate.search.backend.schema_management.minimal_required_status = green
hibernate.search.backend.schema_management.minimal_required_status_wait_timeout = 10000
# To configure a specific index:
hibernate.search.backend.indexes.<index-name>.schema_management.minimal_required_status = green
hibernate.search.backend.indexes.<index-name>.schema_management.minimal_required_status_wait_timeout = 10000
```

- `minimal_required_status` defines the minimal required status of an index before creation is considered complete. The default for this property is `yellow`, except on Amazon OpenSearch Serverless where index status checks are skipped because that platform does not support index status checks.
- `minimal_required_status_wait_timeout` defines the maximum time to wait for this status, as an integer value in milliseconds. The default for this property is `10000`.

These properties are only effective when creating or validating an index as part of schema management.

18.9. Index layout

Hibernate Search works with aliased indexes. This means an index with a given name in Hibernate Search will not directly be mapped to an index with the same name in Elasticsearch.

The index layout is how Hibernate Search index names are mapped to Elasticsearch indexes, and the strategy controlling that layout is set at the backend level:

```
hibernate.search.backend.layout.strategy = simple
```

The default for this property is `simple`.

See the following subsections for details about available strategies.

18.9.1. `simple`: the default, future-proof strategy

For an index whose name in Hibernate Search is `myIndex`:

- If Hibernate Search creates the index automatically, it will name the index `myindex-000001` and will automatically create the write and read aliases.
- Write operations (indexing, purge, …) will target the alias `myindex-write`.
- Read operations (searching, explaining, …) will target the alias `myindex-read`.

The `simple` layout is a bit more complex than it could be, but it follows the best practices.

Using aliases has a significant advantage over directly targeting the index: it makes full reindexing on a live application possible without downtime, which is useful in particular when listener-triggered indexing is disabled (completely or partially) and you need to fully reindex periodically (for example on a daily basis).

With aliases, you just need to direct the read alias (used by search queries) to an old copy of the index, while the write alias (used by document writes) is redirected to a new copy of the index. Without aliases (in particular with the `no-alias` layout), this is impossible.

This "zero-downtime" reindexing, which shares some characteristics with "blue/green" deployment, is not currently provided by Hibernate Search itself. However, you can implement it in your application by directly issuing commands to Elasticsearch's REST APIs. The basic sequence of actions is the following:

1. Create a new index, `myindex-000002`.
2. Switch the write alias, `myindex-write`, from `myindex-000001` to `myindex-000002`.
3. Reindex, for example using the mass indexer.
4. Switch the read alias, `myindex-read`, from `myindex-000001` to `myindex-000002`.
5. Delete `myindex-000001`.

Note this will only work if the Hibernate Search mapping did not change; a zero-downtime upgrade with a changing schema would be considerably more complex. You will find discussions on this topic in HSEARCH-2861 and HSEARCH-3499.

18.9.2. `no-alias`: a strategy without index aliases

This strategy is mostly useful on legacy clusters.

For an index whose name in Hibernate Search is `myIndex`:

- If Hibernate Search creates the index automatically, it will name the index `myindex` and will not create any alias.
- Write operations (indexing, purge, …) will target the index directly by its name, `myindex`.
- Read operations (searching, explaining, …) will target the index directly by its name `myindex`.

18.9.3. Custom strategy

If the built-in layout strategies do not fit your requirements, you can define a custom layout in two simple steps:

1. Define a class that implements the interface `org.hibernate.search.backend.elasticsearch.index.layout.IndexLayoutStrategy`.
2. Configure the backend to use that implementation by setting the configuration property `hibernate.search.backend.layout.strategy` to a bean reference pointing to the implementation, for example `class:com.mycompany.MyLayoutStrategy`.

For example, the implementation below will lead to the following layout for an index named `myIndex`:

- Write operations (indexing, purge, …) will target the alias `myindex-write`.
- Read operations (searching, explaining, …) will target the alias `myindex` (no suffix).
- If Hibernate Search creates the index automatically at exactly 19:19:00 on November 6th, 2017, it will name the index `myindex-20171106-191900-000000000`.

Example 18.9: Implementing a custom index layout strategy with the Elasticsearch backend

```
import java.time.Clock;
import java.time.Instant;
import java.time.ZoneOffset;
```

```
import java.time.format.DateTimeFormatter;
import java.util.Locale;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import org.hibernate.search.backend.elasticsearch.index.layout.IndexLayoutStrategy;

public class CustomLayoutStrategy implements IndexLayoutStrategy {

    private static final DateTimeFormatter INDEX_SUFFIX_FORMATTER =
            DateTimeFormatter.ofPattern( "uuuuMMdd-HHmmss-SSSSSSSSS", Locale.ROOT )
                    .withZone( ZoneOffset.UTC );
    private static final Pattern UNIQUE_KEY_PATTERN =
            Pattern.compile( "(.*)-\\d+-\\d+-\\d+" );

    @Override
    public String createInitialElasticsearchIndexName(String hibernateSearchIndexName) {
        // Clock is Clock.systemUTC() in production, may be overridden in tests
        Clock clock = MyApplicationClock.get();
        return hibernateSearchIndexName + "-"
                + INDEX_SUFFIX_FORMATTER.format( Instant.now( clock ) );
    }

    @Override
    public String createWriteAlias(String hibernateSearchIndexName) {
        return hibernateSearchIndexName + "-write";
    }

    @Override
    public String createReadAlias(String hibernateSearchIndexName) {
        return hibernateSearchIndexName;
    }

    @Override
    public String extractUniqueKeyFromHibernateSearchIndexName(
            String hibernateSearchIndexName) {
        return hibernateSearchIndexName;
    }

    @Override
    public String extractUniqueKeyFromElasticsearchIndexName(
            String elasticsearchIndexName) {
        Matcher matcher = UNIQUE_KEY_PATTERN.matcher( elasticsearchIndexName );
        if ( !matcher.matches() ) {
            throw new IllegalArgumentException(
                    "Unrecognized index name: " + elasticsearchIndexName
            );
        }
        return matcher.group( 1 );
    }
}
```

18.9.4. Retrieving index or alias names

Index or alias names used to read and write can be retrieved from the metamodel.

Example 18.10: Retrieving the index names from an Elasticsearch index manager

```
SearchMapping mapping = /* ... */ ①
IndexManager indexManager = mapping.indexManager( "Book" ); ②
ElasticsearchIndexManager esIndexManager = indexManager.unwrap( ElasticsearchIndexManager.class ); ③
ElasticsearchIndexDescriptor descriptor = esIndexManager.descriptor();④
String readName = descriptor.readName();⑤
String writeName = descriptor.writeName();⑤
```

① Retrieve the `SearchMapping`.

② Retrieve the `IndexManager`.

③ Narrow down the index manager to the `ElasticsearchIndexManager` type.
④ Get the index descriptor.
⑤ Get the index (or alias) read and write names.

18.10. Schema ("mapping")

What Elasticsearch calls the "mapping" is the schema assigned to each index, specifying the data type and capabilities of each "property" (called an "index field" in Hibernate Search).

For the most part, the Elasticsearch mapping is inferred from the mapping configured through Hibernate Search's mapping APIs, which are generic and independent of Elasticsearch.

Aspects that are specific to the Elasticsearch backend are explained in this section.

Hibernate Search can be configured to push the mapping to Elasticsearch when creating the indexes through schema management.

18.10.1. Field types

Available field types

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

Field types that are not in this list can still be used with a little bit more work:

- If a property in the entity model has an unsupported type, but can be converted to a supported type, you will need a bridge. See Binding and bridges.
- If you need an index field with a specific type that is not supported by Hibernate Search, you will need a bridge that defines a native field type. See Index field type DSL extension.

Table 18.1: Field types supported by the Elasticsearch backend

Field type	Data type in Elasticsearch	Limitations
`java.lang.String`	`text` if an analyzer is defined, `keyword` otherwise	-
`java.lang.Byte`	`byte`	-
`java.lang.Short`	`short`	-
`java.lang.Integer`	`integer`	-
`java.lang.Long`	`long`	-
`java.lang.Double`	`double`	-
`java.lang.Float`	`float`	-
`java.lang.Boolean`	`boolean`	-
`java.math.BigDecimal`	`scaled_float` with a `scaling_factor` equal to 10^(`decimalScale`)	-
`java.math.BigInteger`	`scaled_float` with a `scaling_factor` equal to 10^(`decimalScale`)	-
`java.time.Instant`	`date` with format `uuuu-MM-dd'T'HH:mm:ss.SSSSSSSSSZZZZZ`	Lower range/resolution
`java.time.LocalDate`	`date` with format `uuuu-MM-dd`	Lower range/resolution

Field type	Data type in Elasticsearch	Limitations
`java.time.LocalTime`	`date` with format `HH:mm:ss.SSSSSSSSS`	Lower range/resolution
`java.time.LocalDateTime`	`date` with format `uuuu-MM-dd'T'HH:mm:ss.SSSSSSSSS`	Lower range/resolution
`java.time.ZonedDateTime`	`date` with format `uuuu-MM-dd'T'HH:mm:ss.SSSSSSSSSZZZZZ['VV']'`	Lower range/resolution
`java.time.OffsetDateTime`	`date` with format `uuuu-MM-dd'T'HH:mm:ss.SSSSSSSSSZZZZZ`	Lower range/resolution
`java.time.OffsetTime`	`date` with format `HH:mm:ss.SSSSSSSSSZZZZZ`	Lower range/resolution
`java.time.Year`	`date` with format `uuuu`	Lower range/resolution
`java.time.YearMonth`	`date` with format `uuuu-MM`	Lower range/resolution
`java.time.MonthDay`	`date` with format `uuuu-MM-dd`. **The year is always set to 0**.	-
`GeoPoint`	`geo_point`	-

Range and resolution of date/time fields

The Elasticsearch `date` type does not support the whole range of years that can be represented in `java.time` types:

- `java.time` can represent years ranging from `-999.999.999` to `999.999.999`.
- Elasticsearch's `date` type supports dates ranging from year `-292.275.054` to year `292.278.993`.

Values that are out of range will trigger indexing failures.

Resolution is also lower:

- `java.time` supports nanosecond-resolution.
- Elasticsearch's `date` type supports millisecond-resolution.

Precision beyond the millisecond will be lost when indexing.

Index field type DSL extension

Not all Elasticsearch field types have built-in support in Hibernate Search. Unsupported field types can still be used, however, by taking advantage of the "native" field type. Using this field type, the Elasticsearch "mapping" can be defined as JSON directly, giving access to everything Elasticsearch can offer.

Below is an example of how to use the Elasticearch "native" type.

Example 18.11: Using the Elasticearch "native" type

```
public class IpAddressValueBinder implements ValueBinder { ①
    @Override
    public void bind(ValueBindingContext<?> context) {
        context.bridge(
                String.class,
                new IpAddressValueBridge(),
                context.typeFactory() ②
                        .extension( ElasticsearchExtension.get() ) ③
                        .asNative() ④
                                .mapping( "{\"type\": \"ip\"}" ) ⑤
        );
    }

    private static class IpAddressValueBridge implements ValueBridge<String, JsonElement> {
        @Override
        public JsonElement toIndexedValue(String value,
                ValueBridgeToIndexedValueContext context) {
```

```
            return value == null ? null : new JsonPrimitive( value ); ⑥
        }

        @Override
        public String fromIndexedValue(JsonElement value,
                ValueBridgeFromIndexedValueContext context) {
            return value == null ? null : value.getAsString(); ⑦
        }
    }
}
```

① Define a custom binder and its bridge. The "native" type can only be used from a binder, it cannot be used directly with annotation mapping. Here we're defining a value binder, but a type binder, or a property binder would work as well.
② Get the context's type factory.
③ Apply the Elasticsearch extension to the type factory.
④ Call `asNative` to start defining a native type.
⑤ Pass the Elasticsearch mapping as JSON.
⑥ Values of native fields are represented as a `JsonElement` in Hibernate Search. `JsonElement` is a type from the Gson library. Do not forget to format them correctly before you pass them to the backend. Here we are creating a `JsonPrimitive` (a subtype of `JsonElement`) from a `String` because we just need a JSON string, but it's completely possible to handle more complex objects, or even to convert directly from POJOs to JSON using Gson.
⑦ For nicer projections, you can also implement this method to convert from `JsonElement` to the mapped type (here, `String`).

```
@Entity
@Indexed
public class CompanyServer {

    @Id
    @GeneratedValue
    private Integer id;

    @NonStandardField( ①
            valueBinder = @ValueBinderRef(type = IpAddressValueBinder.class) ②
    )
    private String ipAddress;

    // Getters and setters
    // ...

}
```

① Map the property to an index field. Note that value bridges using a non-standard type (such as Elasticsearch's "native" type) must be mapped using the `@NonStandardField` annotation: other annotations such as `@GenericField` will fail.
② Instruct Hibernate Search to use our custom value binder.

18.10.2. Entity type name mapping

When Hibernate Search performs a search query targeting multiple entity types, and thus multiple indexes, it needs to determine the entity type of each search hit in order to map it back to an entity.

There are multiple strategies to handle this "entity type name resolution", and each has pros and cons.

The strategy is set at the backend level:

```
hibernate.search.backend.mapping.type_name.strategy = discriminator
```

The default for this property is `discriminator`.

See the following subsections for details about available strategies.

`discriminator`: type name mapping using a discriminator field

With the `discriminator` strategy, a discriminator field is used to retrieve the entity type name directly from each document.

When indexing, the `_entity_type` field is populated transparently with the name of the entity type for each document.

When searching, the docvalues for the `_entity_type` field are transparently requested from Elasticsearch and extracted from the response.

Pros:

- Works correctly when targeting index aliases.

Cons:

- Small storage overhead: a few bytes of storage per document.
- Requires full reindexing if an entity name changes, even if the index name doesn't change.

`index-name`: type name mapping using the index name

With the `index-name` strategy, the `_index` meta-field returned for each search hit is used to resolve the index name, and from that the entity type name.

Pros:

- No storage overhead.

Cons:

- Relies on the actual index name, not aliases, because the `_index` meta-field returned by Elasticsearch contains the actual index name (e.g. `myindex-000001`), not the alias (e.g. `myindex-read`). Thus, if indexes do not follow the default naming scheme `<hibernateSearchIndexName>-<6 digits>`, a custom index layout must be configured.

18.10.3. Dynamic mapping

By default, Hibernate Search sets the `dynamic` property in Elasticsearch index mappings to `strict`. This means that attempting to index documents with fields that are not present in the mapping will lead to an indexing failure.

If Hibernate Search is the only client, that won't be a problem, since Hibernate Search usually works on declared schema fields only. For the other cases in which we need to change this setting, we can use the following index-level property to change the value.

```
# To configure the defaults for all indexes:
hibernate.search.backend.dynamic_mapping = strict
# To configure a specific index:
hibernate.search.backend.indexes.<index-name>.dynamic_mapping = strict
```

The default for this property is `strict`.

We said that Hibernate Search **usually** works on declared schema fields. More precisely, it always does if no dynamic fields with field templates are defined. When field templates are defined, `dynamic` will be forced to `true`, in order to allow for dynamic fields. In that case, the value of the `dynamic_mapping` property is ignored.

18.10.4. Multi-tenancy

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

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

The multi-tenancy is automatically enabled in the backend if it is enabled in the mapper, e.g. if a multi-tenancy strategy is selected in Hibernate ORM, or if multi-tenancy is explicitly configured in the Standalone POJO mapper.

However, it is possible to enable multi-tenancy manually.

The multi-tenancy strategy is set at the backend level:

```
hibernate.search.backend.multi_tenancy.strategy = none
```

See the following subsections for details about available strategies.

`none`: single-tenancy

The `none` strategy (the default) disables multi-tenancy completely.

Attempting to set a tenant ID will lead to a failure when indexing.

`discriminator`: type name mapping using the index name

With the `discriminator` strategy, all documents from all tenants are stored in the same index. The Elasticsearch ID of each document is set to the concatenation of the tenant ID and original ID.

When indexing, two fields are populated transparently for each document:

- `_tenant_id`: the "discriminator" field holding the tenant ID.
- `_tenant_doc_id`: a field holding the original (tenant-scoped) document ID.

When searching, a filter targeting the tenant ID field is added transparently to the search query to only return search hits for the current tenant. The ID field is used to retrieve the original document IDs.

18.10.5. Custom index mapping

Basics

Hibernate Search can create and validate indexes, but by default created indexes will only include the bare minimum required to index and search: the mapping, and the analysis settings. Should you need to customize some mapping parameters, it is possible to provide a custom mapping to Hibernate Search: it will include the custom mapping when creating an index.

The consistency of the custom Elasticsearch mapping with the Hibernate Search mapping will not get checked in any way. You are responsible for making sure that any override in your mapping can work, e.g. that you're not changing the type of an index field from `text` to `integer`, or disabling `doc_values` on a field used for sorting.

An invalid custom mapping may not trigger any exception on bootstrap, but later while indexing or querying. In the worst case, it could not trigger any exception, but simply lead to incorrect search results. Exercise extreme caution.

```
# To configure the defaults for all indexes:
hibernate.search.backend.schema_management.mapping_file = custom/index-mapping.json
# To configure a specific index:
hibernate.search.backend.indexes.<index-name>.schema_management.mapping_file = custom/index-mapping.json
```

Example 18.12: Possible content of `custom/index-mapping.json` file

```
{
  "properties":{
    "userField":{
      "type":"keyword",
      "index":true,
      "norms":true,
      "doc_values":true
    },
    "userObject":{
      "dynamic":"true",
      "type":"object"
    }
  },
  "_source": {
    "enabled": false
  }
}
```

Properties that are only defined in the custom mappings file but not mapped by Hibernate Search will not be visible to Hibernate Search.

This means Hibernate Search will throw exceptions if you try to reference these properties in the Search DSL, or when writing to a document from a bridge

The file does not need to contain the full mapping: Hibernate Search will automatically inject missing properties (index fields) in the given mapping.

Conflicts between the given mapping and the mapping generated by Hibernate Search will be handled as follows:

1. The `dynamic_templates`/`_routing`/`dynamic` mapping parameters will be those from the given mapping, falling back to the value generated by Hibernate Search (if any).
2. Any other mapping parameters besides the `properties` at the root of the mapping will be those from the given mapping; those generated by Hibernate Search will be ignored.
3. `properties` will be merged, using properties defined in both the given mapping and the mapping generated by Hibernate Search.
4. If a property is defined on both sides, it will be merged recursively, following steps 1-4.

In the example above, the resulting, merged mapping could look like this:

Example 18.13: Possible resulting mapping after merging the content of `custom/index-mapping.json` with the Hibernate Search mapping

```
{
  "_source":{
    "enabled":false
  },
  "dynamic":"strict",
  "properties":{
    "_entity_type":{ ①
      "type":"keyword",
      "index":false
    },
    "title":{ ②
      "type":"text",
      "analyzer":"english"
    },
    "userField":{
      "type":"keyword",
      "norms":true
    },
    "userObject":{
      "type":"object",
      "dynamic":"true"
    }
  }
}
```

① This property is always added by Hibernate Search for internal implementation purposes.
② This is a property generated by Hibernate Search because the indexed entity has a `String name` property annotated with `@FullTextField`.

Disabling `_source`

Using this feature it is possible to disable the `_source` field. For instance, you could pass a `custom/index-mapping.json` file like the following:

Example 18.14: Possible content of `custom/index-mapping.json` file to disable the `_source` field

```
{
  "_source": {
    "enabled": false
  }
}
```

Disabling the `_source` is useful to reduce the size of Elasticsearch indexes on the filesystem, but it comes at a cost.

Several projections rely on the `_source` being enabled. If you try to use projections with `_source` disabled, behavior is undefined: the search query may return `null` hits, or it may fail completely with exceptions.

18.11. Analysis

18.11.1. Basics

Analysis is the text processing performed by analyzers, both when indexing (document processing) and when searching (query processing).

All built-in Elasticsearch analyzers can be used transparently, without any configuration in Hibernate Search: just use their name wherever Hibernate Search expects an analyzer name. However, analysis can also be configured explicitly.

Elasticsearch analysis configuration is not applied immediately on startup: it needs to be pushed to the Elasticsearch cluster.

Hibernate Search will only push the configuration to the cluster if instructed to do so through schema management.

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

1. Define a class that implements the `org.hibernate.search.backend.elasticsearch.analysis.ElasticsearchAnalysisConfigurer` interface.
2. Configure the backend to use that implementation by setting the configuration property `hibernate.search.backend.analysis.configurer` to a bean reference pointing to the implementation, for example `class:com.mycompany.MyAnalysisConfigurer`.

You can pass multiple bean references separated by commas. See Type of configuration properties.

Hibernate Search will call the `configure` method of this implementation on startup, and the configurer will be able to take advantage of a DSL to define analyzers and normalizers.

A different analysis configurer can be assigned to each index:

```
# To set the default configurer for all indexes:
hibernate.search.backend.analysis.configurer = class:com.mycompany.MyAnalysisConfigurer
# To assign a specific configurer to a specific index:
hibernate.search.backend.indexes.<index-name>.analysis.configurer =
class:com.mycompany.MySpecificAnalysisConfigurer
```

If a specific configurer is assigned to an index, the default configurer will be ignored for that index: only definitions from the specific configurer will be taken into account.

18.11.2. Built-in analyzers

Built-in analyzers are available out-of-the-box and don't require explicit configuration. If necessary, they can be overridden by defining your own analyzer with the same name.

The Elasticsearch backend comes with several built-in analyzers. The exact list depends on the version of Elasticsearch and can be found here.

Regardless of the Elasticsearch version, analyzers whose name is listed as a constant in `org.hibernate.search.engine.backend.analysis.AnalyzerNames` are always available:

`default`

: The analyzer used by default with `@FullTextField`.

  This is just an alias for `standard` by default.

`standard`

: Default behavior: first, tokenize using the standard tokenizer, which follows Word Break rules from the Unicode Text Segmentation algorithm, as specified in Unicode Standard Annex #29. Then, lowercase each token.

`simple`

: Default behavior: first, split the text at non-letter characters. Then, lowercase each token.

`whitespace`

: Default behavior: split the text at whitespace characters. Do not change the tokens.

`stop`

: Default behavior: first, split the text at non-letter characters. Then, lowercase each token. Finally, remove English stop words.

`keyword`

: Default behavior: do not change the text in any way.

  With this analyzer a full text field would behave similarly to a keyword field, but with fewer features: no terms aggregations, for example.

Consider using a `@KeywordField` instead.

18.11.3. Built-in normalizers

The Elasticsearch backend does not provide any built-in normalizer.

18.11.4. Custom analyzers and normalizers

The context passed to the configurer exposes a DSL to define analyzers and normalizers:

Example 18.15: Implementing and using an analysis configurer to define analyzers and normalizers with the Elasticsearch backend

```
package org.hibernate.search.documentation.analysis;

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

public class MyElasticsearchAnalysisConfigurer implements ElasticsearchAnalysisConfigurer {
    @Override
    public void configure(ElasticsearchAnalysisConfigurationContext context) {
        context.analyzer( "english" ).custom() ①
                .tokenizer( "standard" ) ②
                .charFilters( "html_strip" ) ③
                .tokenFilters( "lowercase", "snowball_english", "asciifolding" ); ④

        context.tokenFilter( "snowball_english" ) ⑤
                .type( "snowball" )
                .param( "language", "English" ); ⑥

        context.normalizer( "lowercase" ).custom() ⑦
                .tokenFilters( "lowercase", "asciifolding" );

        context.analyzer( "french" ).custom() ⑧
                .tokenizer( "standard" )
                .tokenFilters( "lowercase", "snowball_french", "asciifolding" );

        context.tokenFilter( "snowball_french" )
                .type( "snowball" )
                .param( "language", "French" );
    }
}
```

① Define a custom analyzer named "english", because it will be used to analyze English text such as book titles.
② Set the tokenizer to a standard tokenizer.
③ Set the char filters. Char filters are applied in the order they are given, before the tokenizer.
④ Set the token filters. Token filters are applied in the order they are given, after the tokenizer.
⑤ Note that, for Elasticsearch, any parameterized char filter, tokenizer or token filter must be defined separately and assigned a name.
⑥ Set the value of a parameter for the char filter/tokenizer/token filter being defined.
⑦ Normalizers are defined in a similar way, the only difference being that they cannot use a tokenizer.
⑧ Multiple analyzers/normalizers can be defined in the same configurer.

```
①
hibernate.search.backend.analysis.configurer =
class:org.hibernate.search.documentation.analysis.MyElasticsearchAnalysisConfigurer
```

① Assign the configurer to the backend using a Hibernate Search configuration property.

It is also possible to assign a name to a parameterized built-in analyzer:

Example 18.16: Naming a parameterized built-in analyzer in the Elasticsearch backend

```
context.analyzer( "english_stopwords" ).type( "standard" ) ①
        .param( "stopwords", "_english_" ); ②
```

① Define an analyzer with the given name and type.
② Set the value of a parameter for the analyzer being defined.

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

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

18.11.5. Overriding the default analyzer

The default analyzer when using `@FullTextField` without specifying an analyzer explicitly, is named `default`.

Like any other built-in analyzer, it is possible to override the default analyzer by defining a custom analyzer with the same name:

Example 18.17: Overriding the default analyzer in the Elasticsearch backend

```
package org.hibernate.search.documentation.analysis;

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

public class MyElasticsearchAnalysisConfigurer implements ElasticsearchAnalysisConfigurer {
    @Override
    public void configure(ElasticsearchAnalysisConfigurationContext context) {
        context.analyzer( "english" ).custom() ①
                .tokenizer( "standard" ) ②
                .charFilters( "html_strip" ) ③
                .tokenFilters( "lowercase", "snowball_english", "asciifolding" ); ④

        context.tokenFilter( "snowball_english" ) ⑤
                .type( "snowball" )
                .param( "language", "English" ); ⑥

        context.normalizer( "lowercase" ).custom() ⑦
                .tokenFilters( "lowercase", "asciifolding" );

        context.analyzer( "french" ).custom() ⑧
                .tokenizer( "standard" )
                .tokenFilters( "lowercase", "snowball_french", "asciifolding" );

        context.tokenFilter( "snowball_french" )
                .type( "snowball" )
                .param( "language", "French" );
    }
}
```

① Start the definition of a custom analyzer that happens to be named `default`. Here we rely on constants from `org.hibernate.search.engine.backend.analysis.AnalyzerNames` to use the correct name, but hardcoding `"default"` would work just as well.
② Continue the analyzer definition as we would for any other custom analyzer.

```
①
hibernate.search.backend.analysis.configurer =
class:org.hibernate.search.documentation.analysis.DefaultOverridingElasticsearchAnalysisConfigurer
```

① Assign the configurer to the backend using a Hibernate Search configuration property.

18.12. Custom index settings

Hibernate Search can create and validate indexes, but by default created indexes will only include the bare minimum required to index and search: the mapping, and the analysis settings. Should you need to set some custom index settings, it is possible to provide these settings to Hibernate Search: it will include them when creating an index and take them into account when validating an index.

```
# To configure the defaults for all indexes:
hibernate.search.backend.schema_management.settings_file = custom/index-settings.json
```

```
# To configure a specific index:
hibernate.search.backend.indexes.<index-name>.schema_management.settings_file = custom/index-settings.json
```

Example 18.18: Possible content of `custom/index-settings.json` *file*

```
{
  "number_of_shards": "3",
  "number_of_replicas": "3",
  "analysis": {
    "analyzer": {
      "my_standard-english": {
        "type": "standard",
        "stopwords": "_english_"
      },
      "my_analyzer_ngram": {
        "type": "custom",
        "tokenizer": "my_analyzer_ngram_tokenizer"
      }
    },
    "tokenizer": {
      "my_analyzer_ngram_tokenizer": {
        "type": "ngram",
        "min_gram": "5",
        "max_gram": "6"
      }
    }
  }
}
```

The provided settings will be merged with those generated by Hibernate Search, including analyzer definitions. When analysis is configured both through an analysis configurer and these custom settings, the behavior is undefined; it should not be relied upon.

Custom index setting must be provided in the simplified form, the one without the attribute the `index` attribute.

18.12.1. Max result window size

If the setting `index.max_result_window` is used, Hibernate Search will use this value to limit the returning hits size if no limit has been defined on the query by the user. In this case if Hibernate Search notices there are more results, a warning will be logged.

18.13. Threads

The Elasticsearch backend relies on an internal thread pool to orchestrate indexing requests (add/update/delete) and to schedule request timeouts.

By default, the pool contains exactly as many threads as the number of processors available to the JVM on bootstrap. That can be changed using a configuration property:

```
hibernate.search.backend.thread_pool.size = 4
```

This number is *per backend*, not per index. Adding more indexes will not add more threads.

As all operations happening in this thread-pool are non-blocking, raising its size above the number of processor cores available to the JVM will not bring noticeable performance benefits.

The only reason to alter this setting would be to reduce the number of threads; for example, in an application with a single index with a single indexing queue, running on a machine with 64 processor cores, you might want to bring down the number of threads.

18.14. Indexing queues

Among all the requests sent by Hibernate Search to Elasticsearch, it is expected that there will be a lot of "indexing" requests to create/update/delete a specific document. Sending these requests one by one would be inefficient (mainly because of network latency).

Also, we generally want to preserve the relative order of these requests when they are about the same documents.

For these reasons, Hibernate Search pushes these requests to ordered queues and relies on the Bulk API to send them in batches. Each index maintains 10 queues holding at most 1000 elements each, and each queue will send bulk requests of at most 100 indexing requests. Queues operate independently (in parallel), but each queue sends one bulk request after the other, so at any given time there can be at most 10 bulk requests being sent for each index.

Indexing operations relative to the same document ID are always pushed to the same queue.

It is possible to customize the queues in order to reduce the load on the Elasticsearch server, or on the contrary to improve throughput. This is done through the following configuration properties:

```
# To configure the defaults for all indexes:
hibernate.search.backend.indexing.queue_count = 10
hibernate.search.backend.indexing.queue_size = 1000
hibernate.search.backend.indexing.max_bulk_size = 100
# To configure a specific index:
hibernate.search.backend.indexes.<index-name>.indexing.queue_count = 10
hibernate.search.backend.indexes.<index-name>.indexing.queue_size = 1000
hibernate.search.backend.indexes.<index-name>.indexing.max_bulk_size = 100
```

- `indexing.queue_count` defines the number of queues. Expects a strictly positive integer value. The default for this property is `10`.

  Higher values will lead to more connections being used in parallel, which may lead to higher indexing throughput, but incurs a risk of overloading Elasticsearch, leading to Elasticsearch giving up on some requests and resulting in indexing failures.

- `indexing.queue_size` defines the maximum number of elements each queue can hold. Expects a strictly positive integer value. The default for this property is `1000`.

  Lower values may lead to lower memory usage, especially if there are many queues, but values that are too low will reduce the likeliness of reaching the max bulk size and increase the likeliness of application threads blocking because the queue is full, which may lead to lower indexing throughput.

- `indexing.max_bulk_size` defines the maximum number of indexing requests in each bulk request. Expects a strictly positive integer value. The default for this property is `100`.

  Higher values will lead to more documents being sent in each HTTP request sent to Elasticsearch, which may lead to higher indexing throughput, but incurs a risk of overloading Elasticsearch, leading to Elasticsearch giving up on some requests and resulting in indexing failures.

  Note that raising this number above the queue size has no effect, as bulks cannot include more requests than are contained in the queue.

When a queue is full, any attempt to request indexing will block until the request can be put into the queue.

In order to achieve a reasonable level of performance, be sure to set the size of queues to a high enough number that this kind of blocking only happens when the application is under very high load.

Elasticsearch nodes can only handle so many parallel requests, and in particular they limit the amount of memory available to store all pending requests at any given time.

In order to avoid indexing failures, avoid using overly large numbers for the number of queues and the maximum bulk size, especially if you expect your index to hold large documents.

18.15. Writing and reading

For a preliminary introduction to writing to and reading from indexes in Hibernate Search, including in particular the concepts of *commit* and *refresh*, see Commit and refresh.

18.15.1. Commit

When writing to indexes, Elasticsearch relies on a transaction log to make sure that changes, even uncommitted, are always safe as soon as the REST API call returns.

For that reason, the concept of "commit" is not as important to the Elasticsearch backend, and commit requirements are largely irrelevant.

18.15.2. Refresh

When reading from indexes, Elasticsearch relies on a periodically refreshed index reader, meaning that search queries will return slightly out-of-date results, unless a refresh was forced: this is called near-real-time behavior.

By default, the index reader is refreshed every second, but this can be customized on the Elasticsearch side through index settings: see the `refresh_interval` setting on this page.

18.16. Searching

Searching with the Elasticsearch backend relies on the same APIs as any other backend.

This section details Elasticsearch-specific configuration related to searching.

18.16.1. Scroll timeout

With the Elasticsearch backend, scrolls are subject to timeout. If `next()` is not called for a long period of time (default: 60 seconds), the scroll will be closed automatically and the next call to `next()` will fail.

Use the following configuration property at the backend level to configure the timeout (in seconds):

```
hibernate.search.backend.scroll_timeout = 60
```

The default for this property is `60`.

18.16.2. Partial shard failure

With the Elasticsearch backend, fetching results may result in partial shard failures, i.e. some of the shards will fail to produce results while the others will succeed. In such situations, an Elasticsearch cluster will produce a response with a successful status code, but will contain additional information on failed shards, and the reason they have failed.

By default, Hibernate Search will check if any shards have failed while fetching the results and if so — will throw an exception.

Use the following configuration property at the backend level to change the default behaviour:

```
hibernate.search.backend.query.shard_failure.ignore = true
```

The default for this property is `false`.

Chapter 19. Coordination

19.1. Basics

Coordination is a complex topic, and the problems it solves may appear unclear at first sight. You may find it easier to approach coordination from a higher level.

For a few example architectures involving different coordination strategies and a summary of the differences between each architecture, see Examples of architectures.

For a summary of the differences between coordination strategies specifically in the context of listener-triggered indexing, see Basics.

An application using Hibernate Search usually relies on multiple threads, or even multiple application instances, which will update the database concurrently.

The coordination strategy defines how these threads/nodes will coordinate with each other in order to update indexes according to these database updates, in a way that ensures consistency, prevents data loss, and optimizes performance.

The default strategy provides no coordination. See the following subsections for details about how the default works, as well as how to use other strategies — namely `outbox-polling`.

Coordination strategies are only available for the integration to Hibernate ORM.

See this section for information about coordination in the Standalone POJO mapper.

19.2. No coordination

19.2.1. Basics

The `none` strategy is the simplest and does not involve any additional infrastructure for communication between application nodes.

All explicit or listener-triggered indexing operations are executed directly in application threads, which gives this strategy the unique ability to provide synchronous indexing, at the cost of a few limitations:

- Without coordination, in rare cases, indexing involving `@IndexedEmbedded` may lead to out-of sync indexes
- Without coordination, backend errors during indexing may lead to out-of sync indexes

This strategy is enabled by default, but can also be selected explicitly with the following settings:

```
hibernate.search.coordination.strategy = none
```

Coordination is a complex topic, and the problems it solves may appear unclear at first sight. You may find it easier to approach coordination from a higher level.

For a few example architectures involving different coordination strategies and a summary of the differences between each architecture, see Examples of architectures.

For a summary of the differences between coordination strategies specifically in the context of listener-triggered indexing, see Basics.

19.2.2. How indexing works without coordination

Changes have to occur in the ORM session in order to trigger indexing listeners

See In-session entity change detection and limitations for more details.

Associations must be updated on both sides

See Listener-triggered indexing ignores asymmetric association updates for more details.

Only relevant changes trigger indexing

See Dirty checking for more details.

Entity data is extracted from entities upon session flushes or `SearchSession.close()`

When a Hibernate ORM session is flushed, or (with the Standalone POJO Mapper) when `SearchSession.close()` is called, Hibernate Search will extract data from the entities to build documents to index, and will put these documents in an internal buffer for later

indexing. This extraction may involve loading extra data from the database.

With the Hibernate ORM integration, the fact that this internal buffer is populated on Hibernate ORM session flush means that you can safely `clear()` the session after a `flush()`: entity changes performed up to the flush will be indexed correctly.

If you come from Hibernate Search 5 or earlier, you may see this as a significant improvement: there is no need to call `flushToIndexes()` and update indexes in the middle of a transaction anymore, except for larger volumes of data (see Hibernate ORM and the periodic "flush-clear" pattern with `SearchIndexingPlan`).

However, if you perform a batch process inside a transaction with the Hibernate ORM integration, and call `session.flush()`/`session.clear()` regularly to save memory, be aware that Hibernate Search's internal buffer holding documents to index will grow on each flush, and will not be cleared until the transaction is committed or rolled back. If you encounter memory issues because of that, see Hibernate ORM and the periodic "flush-clear" pattern with `SearchIndexingPlan` for a few solutions.

Extraction of entity data may fetch extra data from the database

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

Hibernate Search tries to only load what is necessary for indexing, but depending on your mapping, this may lead to lazy associations being loaded just to reindex entities, even if you didn't need them in your business code, which may represent an overhead for your application threads as well as your database.

With the Hibernate ORM integration, this extra cost can be mitigated to some extent by:

- leveraging Hibernate ORM's batch fetching: see the `batch_fetch_size` property and the `@BatchSize` annotation.
- leveraging Hibernate ORM's second-level cache, especially for immutable entities referenced from indexed entities (e.g. for reference data such as countries, cities, …).

Indexing is not guaranteed on commit, but only after the application thread returns

When entity changes happen inside a transaction, indexes are not updated immediately, but only after the transaction is successfully committed. That way, if a transaction is rolled back, the indexes will be left in a state consistent with the database, discarding all the index changes that were planned during the transaction.

Similarly, when using the Standalone POJO Mapper, indexes are guaranteed to be updated after `SearchSession.close()` returns.

However, if an error occurs in the backend while indexing, this behavior means that index changes may be lost, leading to out-of-sync indexes. If this is a problem for you, you should consider switching to another coordination strategy.

With the Hibernate ORM integration, when entity changes happen outside any transaction (not recommended), indexes are updated immediately upon session `flush()`. Without that flush, indexes will not be updated automatically.

Index changes may not be visible immediately

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

See Synchronization with the indexes for details.

19.3. `outbox-polling`: additional event tables and polling in background processors

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

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

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

19.3.1. Basics

The `outbox-polling` strategy implements coordination through additional tables in the application database.

Explicit and listener-triggered indexing are implemented by pushing events to an outbox table within the same transaction as the entity changes, and polling this outbox table from background processors which perform indexing.

This strategy is able to provide guarantees that entities will be indexed regardless of temporary I/O errors in backend, at the cost of being only able to perform this indexing asynchronously.

The `outbox-polling` strategy can be enabled with the following settings:

```
hibernate.search.coordination.strategy = outbox-polling
```

If multi-tenancy is enabled, you will need extra configuration.

See Multi-tenancy.

You will also need to add this dependency:

```
<dependency>
   <groupId>org.hibernate.search</groupId>
   <artifactId>hibernate-search-mapper-orm-outbox-polling</artifactId>
   <version>8.4.0.CR1</version>
</dependency>
```

Coordination is a complex topic, and the problems it solves may appear unclear at first sight. You may find it easier to approach coordination from a higher level.

For a few example architectures involving different coordination strategies and a summary of the differences between each architecture, see Examples of architectures.

For a summary of the differences between coordination strategies specifically in the context of listener-triggered indexing, see Basics.

19.3.2. How indexing works with `outbox-polling` coordination

Changes have to occur in the ORM session in order to trigger indexing listeners

See In-session entity change detection and limitations for more details.

Associations must be updated on both sides

See Listener-triggered indexing ignores asymmetric association updates for more details.

Only relevant changes trigger indexing

See Dirty checking for more details.

Indexing happens in a background thread

When a Hibernate ORM session is flushed, Hibernate Search will persist entity change events within the same Hibernate ORM session and the same transaction.

An event processor polls the database for new entity change events, and asynchronously performs reindexing of the appropriate entities when it finds new events (i.e. after the transaction is committed).

The fact that events are persisted on session flush means that you can safely `clear()` the session after a `flush()`: entity changes events detected up to the flush will be persisted correctly.

If you come from Hibernate Search 5 or earlier, you may see this as a significant improvement: there is no need to call `flushToIndexes()` and update indexes in the middle of a transaction anymore.

The background processor will completely reload entities from the database

The background processor responsible for reindexing entities does not have access to the state of the first level cache when the entity change occurred, because it occurred in a different session.

This means each time an entity changes and has to be re-indexed, the background process will load that entity in full. Depending on your mapping, it may also need to load lazy associations to other entities.

This extra cost can be mitigated to some extent by:

- leveraging Hibernate ORM's batch fetching; see the `batch_fetch_size` property and the `@BatchSize` annotation.
- leveraging Hibernate ORM's second-level cache, especially for immutable entities referenced from indexed entities (e.g. for reference data such as countries, cities, …).

Indexing is guaranteed on transaction commit

When entity changes happen inside a transaction, Hibernate Search will persist entity change events within the same transaction.

If the transaction is committed, these events will be committed as well; if it rolls back, the events will be rolled back as well. This guarantees the events will eventually be processed by a background thread and that the indexes will be updated accordingly, but only when (if) the transaction succeeds.

When entity changes happen outside any transaction (not recommended), events indexes are sent immediately after the session `flush()`. Without that flush, indexes will not be updated automatically.

Index changes will not be visible immediately

By default, the application thread will resume after entity change events are committed to the database. This means these changes are safely stored to disk, but this does not mean a search query ran immediately when the thread resumes will take the changes into account: indexing will happen at a later time, asynchronously, in a background processor.

You can configure this event processor to run more often, but it will remain asynchronous.

19.3.3. Impact on the database schema

Basics

The `outbox-polling` coordination strategy needs to store data in additional tables in the application database, so that this data can be consumed by background threads.

This includes in particular an outbox table, to which one row (representing a change event) is pushed every time an entity is changed in a way that requires reindexing.

This also includes an agent table, where Hibernate Search registers every background event processor in order to dynamically assign shards to each application instance, or simply to check that statically assigned shards are consistent.

These tables are accessed through entities that are automatically added to the Hibernate ORM configuration, and as such they should be automatically generated when relying on Hibernate ORM's automatic schema generation.

If you need to integrate the creation/dropping of these tables to your own script, the easiest solution is to have Hibernate ORM generate DDL scripts for your whole schema and copy everything related to constructs (tables, sequences, …) prefixed with `HSEARCH_`. See automatic schema generation, in particular the Hibernate ORM properties `javax.persistence.schema-generation.scripts.action`, `javax.persistence.schema-generation.scripts.create-target` and `javax.persistence.schema-generation.scripts.drop-target`.

Custom schema/table name/etc.

By default, outbox and agent tables, mentioned in the previous section, are expected to be found in the default catalog/schema, and are using uppercased table names prefixed with `HSEARCH_`. Identity generator names used for these tables are prefixed with `HSEARCH_` and suffixed with `_GENERATOR`.

Sometimes there are specific naming conventions for database objects, or a need to separate the domain and technical tables. To allow some flexibility in this area, Hibernate Search provides a set of configuration properties to specify catalog/schema/table names and a custom UUID generator strategy/data type for outbox event and agent tables:

```
# Configure the agent mapping:
hibernate.search.coordination.entity.mapping.agent.catalog=CUSTOM_CATALOG
hibernate.search.coordination.entity.mapping.agent.schema=CUSTOM_SCHEMA
hibernate.search.coordination.entity.mapping.agent.table=CUSTOM_AGENT_TABLE
hibernate.search.coordination.entity.mapping.agent.uuid_gen_strategy=time
hibernate.search.coordination.entity.mapping.agent.uuid_type=BINARY
# Configure the outbox event mapping:
hibernate.search.coordination.entity.mapping.outboxevent.catalog=CUSTOM_CATALOG
hibernate.search.coordination.entity.mapping.outboxevent.schema=CUSTOM_SCHEMA
hibernate.search.coordination.entity.mapping.outboxevent.table=CUSTOM_OUTBOX_TABLE
hibernate.search.coordination.entity.mapping.outboxevent.uuid_gen_strategy=time
hibernate.search.coordination.entity.mapping.outboxevent.uuid_type=BINARY
```

- `agent.catalog` defines the database catalog to use for the agent table.

  Defaults to the default catalog configured in Hibernate ORM.

- `agent.schema` defines the database schema to use for the agent table.

  Defaults to the default schema configured in Hibernate ORM.

- `agent.table` defines the name of the agent table.

  Defaults to `HSEARCH_AGENT`.

- `agent.uuid_gen_strategy` defines name of the UUID generator strategy used for the agent table. Available options are `auto`/`random`/`time`. `auto` is a default and is the same as `random` which uses `UUID#randomUUID()`. `time` is an IP based strategy consistent with IETF RFC 4122.

  Defaults to `auto`.

- `agent.uuid_type` defines the name of the Hibernate `SqlType` used for representing an UUID in the agent table. Hibernate Search provides a special `default` option that is going to be used by default and will result into one of `UUID`/`BINARY`/`CHAR` depending on the database in use. While currently Hibernate Search will use the dialect's default representation of the UUID in the database, it is not a guarantee. If a specific type is required, it is best to provide it explicitly via this property. Please refer to SqlTypes to see the list of available type codes supported by Hibernate ORM. The SQL type code can be passed as a name of a corresponding constant in `org.hibernate.type.SqlTypes` or as an integer value.

  Defaults to `default`.

- `outboxevent.catalog` defines the database catalog to use for the outbox event table.

  Defaults to the default catalog configured in Hibernate ORM.

- `outboxevent.schema` defines the database schema to use for the outbox event table.

  Defaults to the default schema configured in Hibernate ORM.

- `outboxevent.table` defines the name of the outbox events table.

  Defaults to `HSEARCH_OUTBOX_EVENT`.

- `outboxevent.uuid_gen_strategy` defines name of the UUID generator strategy used for the outbox event table. Available options are `auto`/`random`/`time`. `auto` is a default and is the same as `random` which uses `UUID#randomUUID()`. `time` is an IP based strategy consistent with IETF RFC 4122.

  Defaults to `auto`.

- `outboxevent.uuid_type` defines the name of the Hibernate `SqlType` used for representing an UUID in the outbox event table. Hibernate Search provides a special `default` option that is going to be used by default and will result into one of `UUID`/`BINARY`/`CHAR` depending on the database in use. While currently Hibernate Search will use the dialect's default representation of the UUID in the database, it is not a guarantee. If a specific type is required, it is best to provide it explicitly via this property. Please refer to SqlTypes to see the list of available type codes supported by Hibernate ORM. The SQL type code can be passed as a name of a corresponding constant in `org.hibernate.type.SqlTypes` or as an integer value.

  Defaults to `default`.

If your application relies on automatic database schema generation, make sure that the underlying database supports catalogs/schemas when specifying them. Also check if there are any constraints on name length and case sensitivity.

It is not required to provide all properties at the same time. For example, you can customize the schema only. Unspecified properties will use their defaults.

19.3.4. Sharding and pulse

In order to avoid unnecessarily indexing the same entity multiple times on different application nodes, Hibernate Search partitions the entities in what it calls "shards":

- Each entity belongs to exactly one shard.
- Each application node involved in event processing is uniquely assigned one or more shards, and will only process events related to entities in these shards.

In order to reliably assign shards, Hibernate Search adds an agent table to the database, and uses that table to register agents involved in indexing (most of the time, one application instance = one agent = the event processor). The registered agents form a cluster.

To make sure that agents are always assigned one shard, and that one shard is never assigned to more than one agent, each agent will periodically perform a "pulse", updating and inspecting the agent table.

What happens during a pulse depends on the type of agent. During a "pulse":

- An event processor will:
  - update its own entry in the agent table, to let other agents knows it's still active;

- forcibly remove entries from other agents if it detects that these agents expired (did not update their entry for a long time);
- detect and report configuration mistakes if using static sharding, e.g. two agents assigned to the same shard;
- decide to suspend itself if a mass indexer is running;
- trigger rebalancing as necessary if using dynamic sharding; e.g. when it detects that new agents recently joined the cluster, or that agents left the cluster (voluntarily or forcibly).

- A mass indexer will:
  - update its own entry in the agent table, to let other agents knows it's still active;
  - forcibly remove entries from other agents if it detects that these agents expired (did not update their entry for a long time);
  - switch to active waiting mode (frequent polling) if it notices some event processors are still running;
  - switch to pulse-only mode (infrequent polling) and give the green light for mass indexing to start if it notices no event processors are running anymore.

For more details about dynamic and static sharding, see the following sections.

19.3.5. Event processor

Basics

Among agents of the `outbox-polling` coordination strategy executing in the background, the most important one is the event processor: it polls the outbox table for events and then re-indexes the corresponding entities when new events are found.

The event processor can be configured using the following configuration properties:

```
hibernate.search.coordination.event_processor.enabled = true
hibernate.search.coordination.event_processor.polling_interval = 100
hibernate.search.coordination.event_processor.pulse_interval = 2000
hibernate.search.coordination.event_processor.pulse_expiration = 30000
hibernate.search.coordination.event_processor.batch_size = 50
hibernate.search.coordination.event_processor.transaction_timeout = 10
hibernate.search.coordination.event_processor.retry_delay = 15
```

- `event_processor.enabled` defines whether the event processor is enabled, as a boolean value. The default for this property is `true`, but it can be set to `false` to disable event processing on some application nodes, for example to dedicate some nodes to HTTP request processing and other nodes to event processing.
- `event_processor.polling_interval` defines how long to wait for another query to the outbox events table after a query didn't return any event, as an integer value in milliseconds. The default for this property is `100`.

  High values mean higher latency between an entity change and the corresponding update in the index, but less stress on the database when there are no events to process.

  Low values mean lower latency between an entity change and the corresponding update in the index, but more stress on the database when there are no events to process.

- `event_processor.pulse_interval` defines how long the event processor can poll for events before it must perform a "pulse", as an integer value in milliseconds. The default for this property is `2000`.

  See the sharding basics for information about "pulses".

  The pulse interval must be set to a value between the polling interval (see above) and one third (1/3) of the expiration interval (see below).

  Low values (closer to the polling interval) mean less time wasted not processing events when a node joins or leaves the cluster, and reduced risk of wasting time not processing events because an event processor is incorrectly considered disconnected, but more stress on the database because of more frequent checks of the list of agents.

  High values (closer to the expiration interval) mean more time wasted not processing events when a node joins or leaves the cluster, and increased risk of wasting time not processing events because an event processor is incorrectly considered disconnected, but less stress on the database because of less frequent checks of the list of agents.

- `event_processor.pulse_expiration` defines how long an event processor "pulse" remains valid before considering the processor disconnected and forcibly removing it from the cluster, as an integer value in milliseconds. The default for this property is `30000`.

  See the sharding basics for information about "pulses".

  The expiration interval must be set to a value at least 3 times larger than the pulse interval (see above).

  Low values (closer to the pulse interval) mean less time wasted not processing events when a node abruptly leaves the cluster due to a crash or network failure, but increased risk of wasting time not processing events because an event processor is incorrectly considered

disconnected.

High values (much larger than the pulse interval) mean more time wasted not processing events when a node abruptly leaves the cluster due to a crash or network failure, but reduced risk of wasting time not processing events because an event processor is incorrectly considered disconnected.

- `event_processor.batch_size` defines how many outbox events, at most, are processed in a single transaction as an integer value. The default for this property is `50`.

  High values mean a lower number of transactions opened by the background process and may increase performance thanks to the first-level cache (persistence context), but will increase memory usage and in extreme cases may lead to `OutOfMemoryErrors`.

- `event_processor.transaction_timeout` defines the timeout for transactions processing outbox events as an integer value in seconds.

  Only effective when a JTA transaction manager is configured.

  When using JTA and this property is not set, Hibernate Search will use whatever default transaction timeout is configured in the JTA transaction manager.

- `event_processor.retry_delay` defines how long the event processor must wait before re-processing an event after its processing failed, as a positive integer value in seconds. The default for this property is `30`.

  Use the value `0` to reprocess failed events as soon as possible, with no delay.

Sharding

By default, sharding is dynamic: Hibernate Search registers each application instance in the database, and uses that information to dynamically assign a single, unique shard to each application instance, updating assignments as instances start or stop. Dynamic sharding does not accept any configuration beyond the basics.

If you want to configure sharding explicitly, you can use static sharding by setting the following configuration properties:

```
hibernate.search.coordination.event_processor.shards.total_count = 4
hibernate.search.coordination.event_processor.shards.assigned = 0
```

- `shards.total_count` defines the total number of shards as an integer value. This property has no default and must be set explicitly if you want static sharding. It must be set to the same value on all application nodes with assigned shards. When this property is set, `shards.assigned` must also be set
- `shards.assigned` defines the shards assigned to the application node as an integer value, or multiple comma-separated integer values. This property has no default and must be set explicitly if you want static sharding. When this property is set, `shards.total_count` must also be set.

  Shards are referred to by an index in the range `[0, total_count - 1]` (see above for `total_count`). A given application node must be assigned at least one shard but may be assigned multiple shards by setting `shards.assigned` to a comma-separated list, e.g. `0,3`.

Each shard must be assigned to one and only one application node.

Event processing simply won't start until every shard has exactly one node.

Example 19.1: Example of static sharding settings

For example, the following configuration with 4 application nodes would assign shard `0` to application node #0, shard `1` application node #1, and no shard at all to application nodes #2 and #3:

```
# Node #0
hibernate.search.coordination.strategy = outbox-polling
hibernate.search.coordination.event_processor.shards.total_count = 2
hibernate.search.coordination.event_processor.shards.assigned = 0
```

```
# Node #1
hibernate.search.coordination.strategy = outbox-polling
hibernate.search.coordination.event_processor.shards.total_count = 2
hibernate.search.coordination.event_processor.shards.assigned = 1
```

```
# Node #2
hibernate.search.coordination.strategy = outbox-polling
hibernate.search.coordination.event_processor.enabled = false
```

```
# Node #3
hibernate.search.coordination.strategy = outbox-polling
hibernate.search.coordination.event_processor.enabled = false
```

Processing order

The order in which outbox events are processed can be tuned with a configuration property:

```
hibernate.search.coordination.event_processor.order = auto
```

Available options are:

`auto`

The default value.

Picks the safest, most appropriate order based on the dialect and other settings:

- When using time-based UUIDs for outbox events (see Impact on the database schema), picks `id`.
- Otherwise, if using a Microsoft SQL Server dialect, picks `none`.
- Otherwise, picks `time`.

`none`

Process outbox events in no particular order.

This essentially means events will be consumed in a database-specific, undetermined order.

In setups with multiple event processors, this reduces the rate of background failures caused by transaction deadlocks (in particular with Microsoft SQL Server), which does not technically "fix" event processing (those failures are handled automatically by trying again anyway), but may improve performance and reduce unnecessary noise in logs.

However, this may lead to situations where the processing of one particular event is continuously postponed due to newer events being processed before that particular event, which can be a problem in write-intensive scenarios where the event queue is never empty.

`time`

Process outbox events in "time" order, i.e. in the order events are created.

This ensures events are processed more or less in the order they were created and avoids situations where the processing of one particular event is continuously postponed due to newer events being processed before that particular event.

However, in setups with multiple event processors, this may increase the rate of background failures caused by transaction deadlocks (in particular with Microsoft SQL Server), which does not technically break event processing (those failures are handled automatically by trying again anyway), but may reduce performance and lead to unnecessary noise in logs.

`id`

Process outbox events in identifier order.

If outbox event identifiers are time-based UUIDs (see Impact on the database schema), this behaves similarly to `time`, but without the risk of deadlocks.

If outbox event identifiers are random UUIDs (see Impact on the database schema), this behaves similarly to `none`.

19.3.6. Mass indexer

Basics

During mass indexing, an application instance will exceptionally bypass sharding and index entities from any shard.

Bypassing sharding can be dangerous, because indexing the same entity simultaneously from an event processor and the mass indexer could potentially result in an out-of-sync index in some rare situations. This is why, to be perfectly safe, event processing gets suspended while mass indexing is in progress. Events are still produced and persisted, but their processing gets delayed until mass indexing finishes.

The suspension of event processing is achieved by registering a mass indexer agent in the agent table, which event processors will eventually detect and react to by suspending themselves. When mass indexing finishes, the mass indexer agent is removed from the agent table, event processors detect that and resume event processing.

The mass indexer agent can be configured using the following configuration properties:

```
hibernate.search.coordination.mass_indexer.polling_interval = 100
hibernate.search.coordination.mass_indexer.pulse_interval = 2000
hibernate.search.coordination.mass_indexer.pulse_expiration = 30000
```

- `mass_indexer.polling_interval` defines how long to wait for another query to the agent table when actively waiting for event processors to suspend themselves, as an integer value in milliseconds. The default for this property is `100`.

  Low values will reduce the time it takes for the mass indexer agent to detect that event processors finally suspended themselves, but will increase the stress on the database while the mass indexer agent is actively waiting.

  High values will increase the time it takes for the mass indexer agent to detect that event processors finally suspended themselves, but will reduce the stress on the database while the mass indexer agent is actively waiting.

- `mass_indexer.pulse_interval` defines how long the mass indexer can wait before it must perform a "pulse", as an integer value in milliseconds. The default for this property is `2000`.

  See the sharding basics for information about "pulses".

  The pulse interval must be set to a value between the polling interval (see above) and one third (1/3) of the expiration interval (see below).

  Low values (closer to the polling interval) mean reduced risk of event processors starting to process events again during mass indexing because a mass indexer agent is incorrectly considered disconnected, but more stress on the database because of more frequent updates of the mass indexer agent's entry in the agent table.

  High values (closer to the expiration interval) mean increased risk of event processors starting to process events again during mass indexing because a mass indexer agent is incorrectly considered disconnected, but less stress on the database because of less frequent updates of the mass indexer agent's entry in the agent table.

- `mass_indexer.pulse_expiration` defines how long an event processor "pulse" remains valid before considering the processor disconnected and forcibly removing it from the cluster, as an integer value in milliseconds. The default for this property is `30000`.

  See the sharding basics for information about "pulses".

  The expiration interval must be set to a value at least 3 times larger than the pulse interval (see above).

  Low values (closer to the pulse interval) mean less time wasted with event processors not processing events when a mass indexer agent terminates due to a crash, but increased risk of event processors starting to process events again during mass indexing because a mass indexer agent is incorrectly considered disconnected.

  High values (much larger than the pulse interval) mean more time wasted with event processors not processing events when a mass indexer agent terminates due to a crash, but reduced risk of event processors starting to process events again during mass indexing because a mass indexer agent is incorrectly considered disconnected.

19.3.7. Multi-tenancy

If you use Hibernate ORM's multi-tenancy support, you will need to configure the list of all possible tenant identifiers.

Failing to mention a tenant identifier in this configuration might result in events piling up in the outbox table without ever being processed, or in exceptions being thrown upon mass indexing due to incomplete configuration.

Apart from that, multi-tenancy support should be fairly transparent: Hibernate Search will simply duplicate event processors for each tenant identifiers.

You can use different configuration for different tenants by using a different root for configuration properties:

- `hibernate.search.coordination` is the default root, whose properties will be used as a default for all tenants.
- `hibernate.search.coordination.tenants.<tenant-identifier>` is the tenant-specific root.

See below for an example.

Example 19.2: Configuration of coordination in a multi-tenant application with a node dedicated to a single tenant

Node 1:

```
hibernate.search.multi_tenancy.tenant_ids=tenant1,tenant2,tenant3,tenant4
hibernate.search.coordination.strategy = outbox-polling
hibernate.search.coordination.tenants.tenant1.event_processor.enabled = false ①
```

① This node will process events for all tenants **except** `tenant1`.

Node 2:

```
hibernate.search.multi_tenancy.tenant_ids=tenant1,tenant2,tenant3,tenant4
hibernate.search.coordination.strategy = outbox-polling
hibernate.search.coordination.event_processor.enabled = false ①
hibernate.search.coordination.tenants.tenant1.event_processor.enabled = true ②
```

① This node will not process events for any tenant…

② … **except** `tenant1`.

19.3.8. Aborted events

If something goes wrong when an outbox event is processed, the event processor will try to re-process the events two times, after that the event will be marked as aborted. Aborted events won't be processed by the processor.

Hibernate Search exposes some APIs to work on aborted events.

Example 19.3: Use the API for the aborted events

```
OutboxPollingSearchMapping searchMapping = Search.mapping( sessionFactory ).extension( OutboxPollingExtension
.get() ); ①
long count = searchMapping.countAbortedEvents(); ②
searchMapping.reprocessAbortedEvents(); ③
searchMapping.clearAllAbortedEvents(); ④
```

① To have the access to the API we need to pass the `OutboxPollingExtension`

② Count aborted events

③ Re-process aborted events

④ Clear all the aborted events

If multi-tenancy is enabled, it will be necessary to pass the tenant id to target the tenant operations are performed on.

Example 19.4: Use the API for the aborted events with multi-tenancy

```
long count = searchMapping.countAbortedEvents( tenantId ); ①
searchMapping.reprocessAbortedEvents( tenantId ); ②
searchMapping.clearAllAbortedEvents( tenantId ); ③
```

① Count aborted events for the given tenantId

② Re-process aborted events for the given tenantId

③ Clear all the aborted events for the given tenantId

Chapter 20. Static metamodel

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

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

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

Hibernate Search's static metamodel is a set of generated classes that represents the structure of each entity's index, thereby allowing type-safe references to index fields when creating search queries through the Search DSL.

The basic principles are very similar to the JPA static metamodel available in Hibernate ORM, but in the case of Search the metamodel is about indexes rather than entities.

20.1. Overview of the static metamodel

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

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

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

The static metamodel class describes the index structure. Each indexed entity (index) is represented by a single class that may contain inner classes describing object fields (e.g. from indexed-embedded properties). These classes contain field references representing index fields and their search capabilities.

The name of this root class is constructed from the indexed entity name by adding `__` (two underscores) suffix, e.g. `MySearchEntity__`. If the indexed entity is a (static) inner class, then all the owning classes will be part of the name delimited with a `_` (single underscore), e.g. `MyOuterClass_MySearchEntity__`. These classes are created in the same package where the search entity they represent is located.

The root metamodel class that describes the indexed entity has a static field `INDEX` that you can use to interact with the metamodel. It serves two primary purposes:

- It simplifies creating the search scope and building search queries for such scope.
- It provides a way to reference index fields when constructing queries.

While the default naming convention was described in this section, there are plans to provide more flexibility in this area to the users (see HSEARCH-5366) .

```
SearchSession searchSession = /* ... */ ①
var scope = Book__.INDEX.scope( searchSession ); ②

List<Book> hits = searchSession.search( scope )
        .where( f -> f.match()
                .field( Book__.INDEX.title ).field( Book__.INDEX.description ) ③
                .matching( "robot" ) )
        .fetchHits( 20 );
```

① Obtain the search session.
② Create the search scope over the book index. Using such scope in the Search DSL will automatically limit acceptable field references to the ones obtained from the `Book__.INDEX`
③ Use the metamodel to reference the fields when creating the queries.

20.1.1. Field reference types

A field reference type describes the set of search capabilities a particular index field has, in particular, what kind of projections/aggregations/predicates/sorts are allowed, if any. It does so by implementing a subset of search "trait" interfaces defined in `org.hibernate.search.engine.search.reference.*`, which in turn allows performing compile-time checks when building search queries.

```
List<String> titles = searchSession.search( scope )
        .select( f -> f.field( Book__.INDEX.title ) ) ①
        .where( f -> f.matchAll() )
        .fetchHits( 20 );
```

① If the title field is projectable then this compiles fine, otherwise compilation fails with an error similar to:

```
error: no suitable method found for
field(ValueFieldReferenceP0P13P14P2P4P5P6P7P8P9<Book__,String,String,String,String>)
```

as such field reference (for a filed with `Projectable.NO`) will not implement the required `FieldProjectionFieldReference` interface.

The field reference also provides a way to quickly switch between different value models when necessary.

```
List<Book> hits = searchSession.search( scope )
        .where( f -> f.match()
                .field( Book__.INDEX.genre.string() ) ①
                .matching( "CRIME_FICTION" ) ) ②
        .fetchHits( 20 );
```

① Calling `string()` on a field reference is an equivalent to `.field( "genre" ).matching( <search value>, ValueModel.STRING )`

20.2. Annotation processor

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

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

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

20.2.1. Enabling the annotation processor

Hibernate Search provides a dedicated annotation processor to generate the static metamodel classes. This annotation processor is located in the `org.hibernate.search:hibernate-search-processor`.

The annotation processor has to be added to the build, e.g. for Maven:

```
<plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <executions>
        <execution>
            <id>default-compile</id>
            <configuration>
                <annotationProcessorPathsUseDepMgmt>true</annotationProcessorPathsUseDepMgmt> ①
                <annotationProcessors>
                    <annotationProcessor>
org.hibernate.search.processor.HibernateSearchProcessor</annotationProcessor> ②
                </annotationProcessors>
                <annotationProcessorPaths>
                    <path>
                        <groupId>org.hibernate.search</groupId>
                        <artifactId>hibernate-search-processor</artifactId> ③
                    </path>
                    <path>
                        <groupId>org.hibernate.search</groupId>
                        <artifactId>hibernate-search-backend-lucene</artifactId> ④
                    </path>
                </annotationProcessorPaths>
            </configuration>
        </execution>
```

```
        </executions>
    </plugin>
```

① This setting is required for the annotation processor dependencies to be managed by the dependency management section. Particularly useful when using Hibernate Search BOM. More information is available in Maven compiler plugin documentation.

② Provide the fully qualified class name of the annotation processor that generates the metamodel.

③ Add the `org.hibernate.search:hibernate-search-processor` dependency to the annotation processor path (a superset of the compile path), so the Java compiler can find the processor.

④ Add the backend dependency, in this example, the Lucene backend, to the annotation processor path. It is important to include the same backend that the application is using to make sure that the generated metamodel classes reflect all the backend specifics. For example, backends might have different defaults, resulting in a different set of search traits per specific field, depending on the backend.

The version of both annotation processor and backend dependencies can be ommitted in the definition of the annotation paths, because they are defined in the Hibernate Search BOM, which we recommend you import via dependency management. This way the generated metamodel classes will be based on the same backend that the application uses.

20.2.2. Configuration

The annotation processor options are passed as the compiler arguments with the `-A` key:

```
<plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <executions>
        <execution>
            <id>default-compile</id>
            <configuration>
                <compilerArgs>
                    <arg>-Aorg.hibernate.search.processor.generated_annotation.timestamp=false</arg> ①
                    <arg><!-- ... --></arg> ②
                </compilerArgs>
                <!-- ... --> ③
            </configuration>
        </execution>
    </executions>
</plugin>
```

① Pass the annotation processor parameters using the `-A` key.

② Pass any other compiler arguments required by the build.

③ Further compiler plugin configuration.

The following annotation processor configuration properties are available:

`org.hibernate.search.processor.generated_annotation.add`

Description

Whether to add the `@Generated` annotation to the generated static metamodel classes.

Default value

`true`

`org.hibernate.search.processor.generated_annotation.timestamp`

Description

Defines whether the `@Generated` annotation includes the `date` attribute. Having the date attribute will result in non-reproducible builds, as the timestamp will be different for each compilation. Hence, it is disabled by default.

Default value

`false`

`org.hibernate.search.processor.backend.version`

Description

Explicitly define the backend version. By default, the processor will use the latest compatible version of the backend. This option can be used if the static metamodel is required for an older backend version. While this option is mostly for the Elasticsearch backend, where it translates into `hibernate.search.backend.version`, it will also translate into `hibernate.search.backend.lucene_version` if the Lucene backend is used.

Default value

`<latest>`

20.2.3. Current annotation processor limitations

While the annotation processor is very much functional and we encourage you to try it, it is still in the early stages of development, and thus has a few limitations:

- Any use of custom binders will be ignored and should produce a compiler warning. This means that if the search entities rely on custom binders, fields that those binders produce will be missing from the generated metamodel.
- Custom mapping annotations are ignored without warning.
- Programmatic mapping is also unsupported by the annotation processor. As the annotation processor cannot possibly execute programmatic mapping defined in the code that is being compiled, this limitation is here to stay.

Chapter 21. Standards and integrations

21.1. Jakarta EE

Hibernate Search targets Jakarta EE where relevant, in particular Jakarta Persistence 3.2 with the Hibernate ORM Mapper.

21.2. Java EE

Hibernate Search no longer supports Java EE.

Use Jakarta EE instead.

21.3. Hibernate ORM 6/5

The Hibernate Search 8.4 series is no longer compatible with Hibernate ORM 6 and 5. It targets the Hibernate ORM 7 series. Check the compatibility matrix to find versions of Hibernate Search compatible with these older versions of Hibernate ORM.

Use Hibernate ORM 7 instead.

21.4. Lucene 10

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

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

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

Hibernate Search 8.4 provides two variants of the Lucene backend.

The default, stable one, that works with JDK 17+ and uses Lucene 9.12.3:

```
<dependency>
    <groupId>org.hibernate.search</groupId>
    <artifactId>hibernate-search-backend-lucene</artifactId>
</dependency>
```

And an incubating one based on Lucene 10.4.0, that requires JDK 21+:

```
<dependency>
    <groupId>org.hibernate.search</groupId>
    <artifactId>hibernate-search-backend-lucene-next</artifactId>
</dependency>
```

`hibernate-search-backend-lucene-next` is tested against the same test suite as `hibernate-search-backend-lucene` and currently provides the same search capabilities. The main difference is the Lucene version backing it. While Hibernate Search 8.4 still targets JDK 17, Lucene, starting with version 10, is leveraging new Java APIs, particularly to work with memory, that are available starting with JDK 21. For users running their applications on JDK 21+ and wanting to get the latest Lucene improvements, it would be beneficial to use this new backend.

Chapter 22. Known issues and limitations

22.1. Without coordination, in rare cases, indexing involving `@IndexedEmbedded` may lead to out-of sync indexes

22.1.1. Description

With the default settings (no coordination), if two entity instances are indexed-embedded in the same "index-embedding" entity, and these two entity instance are updated in parallel transactions, there is a small risk that the transaction commits happen in just the wrong way, leading to the index-embedding entity being re-indexed with only part of the updates.

For example, consider indexed entity A, which index-embeds B and C. The following course of events involving two parallel transactions (T1 and T2) will lead to an out of date index:

- T1: Load B.
- T1: Change B in a way that will require reindexing A.
- T2: Load C.
- T2: Change C in a way that will require reindexing A.
- T2: Request the transaction commit. Hibernate Search builds the document for A. While doing so, it automatically loads B. B appears unmodified, as T1 wasn't committed yet.
- T1: Request the transaction commit. Hibernate Search builds documents to index. While doing so, it automatically loads C. C appears unmodified, as T2 wasn't committed yet.
- T1: Transaction is committed. Hibernate Search automatically sends the updated A to the index. In this version, B is updated, but C is not.
- T2: Transaction is committed. Hibernate Search automatically sends the updated A to the index. In this version, C is updated, but B is not.

This chain of events ends with the index containing a version of A where C is updated, but B is not.

22.1.2. Solutions and workarounds

The following solutions can help circumvent this limitation:

1. Use a safer coordination strategy, e.g. the `outbox-polling` coordination strategy. See in particular Examples of architectures.
2. OR avoid parallel updates to entities that are indexed-embedded in the same indexed entity. This is only possible in very specific setups.
3. OR schedule a full reindexing of your database periodically (e.g. every night) to get the index back in sync with the database.

22.1.3. Roadmap

This limitation is caused directly by the lack of coordination between threads or application nodes, so it can only be addressed completely by configuring coordination.

There are no other solutions currently on the roadmap.

22.2. Without coordination, backend errors during indexing may lead to out-of sync indexes

22.2.1. Description

With the default settings (no coordination), indexing will actually apply index changes in the backend **just after** the transaction commit, without any kind of transaction log for the index changes.

Consequently, should an error occur in the backend while indexing (i.e. an I/O error), this indexing will be cancelled, with no way to cancel the corresponding database transaction: the index will thus become out of sync.

The risk is exclusively related to errors in the backend, mostly to filesystem or network issues. Errors occurring in user code (getters, custom bridges, …) will safely cancel the whole database transaction without indexing anything, ensuring that indexes are still in sync with the database.

22.2.2. Solutions and workarounds

The following solutions can help circumvent this limitation:

1. Use a safer coordination strategy, e.g. the `outbox-polling` coordination strategy. See in particular Examples of architectures.
2. OR schedule a full reindexing of your database periodically (e.g. every night) to get the index back in sync with the database.

22.2.3. Roadmap

This limitation is caused directly by the lack of persistence of entity change events, so it can only be addressed completely by persisting those events, e.g. by switching to the `outbox-polling` coordination strategy.

Some incomplete countermeasures may be considered in future versions, such as automatic in-thread retries, but they will never solve the problem completely, and they are not currently on the roadmap.

22.3. Listener-triggered indexing only considers changes applied directly to entity instances in Hibernate ORM sessions

22.3.1. Description

Due to how Hibernate Search uses internal events of Hibernate ORM in order to detect changes, it will not detect changes resulting from `insert`/`delete`/`update` queries, be it SQL or JPQL/HQL queries.

This is because queries are executed on the database side, without Hibernate ORM or Search having any knowledge of which entities are actually created, deleted or updated.

22.3.2. Solutions and workarounds

One workaround is to reindex explicitly after you run JPQL/SQL queries, either using the `MassIndexer`, using the Jakarta Batch mass indexing job, or explicitly.

22.3.3. Roadmap

One solution to actually detect these changes would be to source entity change events directly from the database, using for example Debezium.

This is tracked as HSEARCH-3513, but is long-term goal.

22.4. Listener-triggered indexing ignores asymmetric association updates

22.4.1. Description

Hibernate ORM is able to handle asymmetric updates of associations, where only the owning side of association is updated and the other side is ignored. The entities in the session will be inconsistent for the duration of the session, but upon reloading they will be consistent once again, due to how entity loading works.

This practice of asymmetric updates of associations can cause problems in applications in general, but also in Hibernate Search specifically, where it may lead to out-of-sync indexes. Thus, it must be avoided.

For example, let's assume an indexed entity `A` has an `@IndexedEmbedded` association `A.b` to entity `B`, and that `B` owns that association on its side, `B.a`. One can just set `B.a` to `null` in order to remove the association between `A` and `B`, and the effect on the database will be exactly what we want.

However, Hibernate Search will only be able to detect that `B.a` changed, and by the time it tries to infer which entities need to be re-indexed, it will no longer be able to know what `B.a` used to refer to. That change in itself is useless to Hibernate Search: Hibernate Search will not know that `A`, specifically, needs to be re-indexed. It will "forget" to reindex `A`, leading to an out-of-sync index where `A.b` still contains `B`.

In the end, the only way for Hibernate Search to know that `A` needs to be re-indexed is to also set `A.b` to `null`, which will cause Hibernate Search to detect that `A.b` changed, and thus that `A` changed too.

22.4.2. Solutions and workarounds

The following solutions can help circumvent this limitation:

1. When you update one side of an association, always update the other side consistently.
2. When the above is not possible, reindex affected entities explicitly after the association update, either using the `MassIndexer`, using the Jakarta Batch mass indexing job, or explicitly.

22.4.3. Roadmap

Hibernate Search may handle asymmetric association updates in the future, by keeping tracks of entities added to / removed from an association. However, this will only solve the problem completely if indexing happens asynchronously in a background thread, such as with the `outbox-polling` coordination strategy. This is tracked as HSEARCH-3567.

Alternatively, sourcing entity change events directly from the database, using for example Debezium, would also solve the problem. This is tracked as HSEARCH-3513, but is long-term goal.

22.5. Listener-triggered indexing is not compatible with `Session` serialization

22.5.1. Description

When listener-triggered indexing is enabled, Hibernate Search collects entity change events to build an "indexing plan" inside the ORM `EntityManager`/`Session`. The indexing plan holds information relative to which entities need to be re-indexed, and sometimes documents that have not been indexed yet.

The indexing plan cannot be serialized.

If the ORM `Session` gets serialized, all collected change events will be lost upon deserializing the session, and Hibernate Search will likely "forget" to reindex some entities.

This is fine in most applications, since they do not rely on serializing the session, but it might be a problem with some JEE applications relying on Bean Passivation.

22.5.2. Solutions and workarounds

Avoid serializing an ORM `EntityManager`/`Session` after changing entities.

22.5.3. Roadmap

There are no plans to address this limitation. We do not intend to support `Session` serialization when Hibernate Search is enabled.

Chapter 23. Troubleshooting

23.1. Finding out what is executed under the hood

For search queries, you can get a human-readable representation of a `SearchQuery` object by calling `toString()` or `queryString()`. Alternatively, rely on the logs: `org.hibernate.search.query` will log every query at the `TRACE` level before it is executed.

For more general information about what is being executed, rely on loggers:

- `org.hibernate.search.lucene.infostream` for Lucene.
- `org.hibernate.search.elasticsearch.client.request` for Elasticsearch.

23.2. Loggers

Here are a few loggers that can be useful when debugging an application that uses Hibernate Search:

`org.hibernate.search.query`

Available for all backends.

Logs every single search query at the `TRACE` level, before its execution.

`org.hibernate.search.elasticsearch.client.request`

Available for Elasticsearch backends only.

Logs requests sent to Elasticsearch at the `DEBUG` or `TRACE` level after their execution. All available request and response information is logged: method, path, execution time, status code, but also the full request and response.

Use the `DEBUG` level to only log non-success requests (status code different from `2xx`), or the `TRACE` level to log every single request.

You can enable pretty-printing (line breaks and indentation) for the request and response using a configuration property.

`org.hibernate.search.lucene.infostream`

Available for Lucene backends only.

Logs low level trace information about Lucene's internal components, at the `TRACE` level.

Enabling the `TRACE` level on this logger is not enough: you must also enable the infostream using a configuration property.

23.3. Frequently asked questions

23.3.1. Unexpected or missing documents in search hits

When some documents unexpectedly match or don't match a query, you will need information about the exact query being executed, and about the index content.

To find out what the query being executed looks like exactly, see Finding out what is executed under the hood.

To inspect the content of the index:

- With the Elasticsearch backend, run simpler queries using either Hibernate Search or the REST APIs directly.
- With the Lucene backend, run simpler queries using Hibernate Search or use the Luke tool distributed as part of the Lucene binary packages.

23.3.2. Unsatisfying order of search hits when sorting by score

When sorting by score, if the documents don't appear in the order you expect, it means some documents have a score that is higher or lower than what you want.

The best way to gain insight into these scores is to just ask the backend to explain how the score was computed. The returned explanation will include a human-readable description of how the score of a specific document was computed.

Regardless of the API used, explanations are rather costly performance-wise: only use them for debugging purposes.

There are two ways to retrieve explanations:

- If you are interested in a particular entity and know its identifier: use the `explain(…)` method on the query. See `explain(…)`: Explaining scores.

- If you just want an explanation for all the top hits: use an `explanation` projection. See here for Lucene and here for Elasticsearch.

23.3.3. Search query execution takes too long

When the execution of a search query is too long, you may need more information about how long it took exactly, and what was executed exactly.

To find out how long a query execution took, use the `took()` method on the search result.

To find out what the query being executed looks like exactly, see Finding out what is executed under the hood.

Chapter 24. Further reading

24.1. Hibernate Search

The Hibernate Search website is a good first stop when looking for information about Hibernate Search, be it to know more about releases, documentation for all versions, migration guides, the roadmap, or simply getting links to the source code and issue tracker.

The Hibernate Search website also includes links to various external resources such as blog posts and talks.

To contribute to, or ask questions about, Hibernate Search or any Hibernate projects, start from the community page on the same website.

Finally, for examples of using Hibernate Search in applications, see:

- The Quarkus quickstart, a sample application using Hibernate Search with the Quarkus framework.
- The "Library" showcase, a sample application using Hibernate Search with the Spring Boot framework.
- The blog posts and talks mentioned above, most of which include tutorials and/or simple applications.

24.2. Elasticsearch

Elasticsearch's reference documentation is an excellent starting point to learn more about Elasticsearch.

24.3. Lucene

To learn more about Lucene, you can get a copy to Lucene in Action (Second Edition), though it covers an older version of Lucene.

Otherwise, you can always try your luck with Lucene's Javadoc.

24.4. Hibernate ORM

If you want to deepen your knowledge of Hibernate ORM, start with the online documentation or get hold of a copy of Java Persistence with Hibernate, Second Edition.

24.5. Other

If you have any further questions or feedback regarding Hibernate Search, reach out to the Hibernate community. We are looking forward to hearing from you.

In case you would like to report a bug use the Hibernate Search JIRA instance.

Chapter 25. Credits

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

The following contributors have been involved in this documentation:

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

Appendix A: List of all available configuration properties

A.1. Hibernate Search Engine

`hibernate.search.background_failure_handler`

The `FailureHandler` instance that should be notified of any failure occurring in a background process (mainly index operations).

Expects a reference to a bean of type `FailureHandler`.

Defaults to `EngineSettings.Defaults.BACKGROUND_FAILURE_HANDLER`, a logging handler.

`hibernate.search.configuration_property_checking.strategy`

How to report the results of configuration property checking.

Configuration property checking will detect an configuration property that is never used, which might indicate a configuration issue.

Expects a `ConfigurationPropertyCheckingStrategyName` value, or a String representation of such value.

Defaults to `EngineSettings.Defaults.CONFIGURATION_PROPERTY_CHECKING_STRATEGY`.

`hibernate.search.backend.type`

The type of the backend.

Only useful if you have more than one backend technology in the classpath; otherwise the backend type is automatically detected.

Expects a String, such as "lucene" or "elasticsearch". See the documentation of your backend to find the appropriate value.

Defaults:

- If there is only one backend type in the classpath, defaults to that backend.
- Otherwise, no default: this property must be set.

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.type`

A.2. Hibernate Search Backend - Lucene

`hibernate.search.backend.analysis.configurer`

The configurer for analysis.

Expects a single-valued or multi-valued reference to beans of type `LuceneAnalysisConfigurer`.

Defaults to no value.

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.analysis.configurer`

`hibernate.search.backend.directory.filesystem_access.strategy`

How to access the filesystem in the directory.

Only available for the "local-filesystem" directory type.

Expects a `FileSystemAccessStrategyName` value, or a String representation of such value.

Defaults to `LuceneIndexSettings.Defaults.DIRECTORY_FILESYSTEM_ACCESS_STRATEGY`.

▼ *Variants of this configuration property*

- `hibernate.search.backend.indexes.<index-name>.directory.filesystem_access.strategy`
- `hibernate.search.backends.<backend-name>.directory.filesystem_access.strategy`
- `hibernate.search.backends.<backend-name>.indexes.<index-name>.directory.filesystem_access.strategy`

`hibernate.search.backend.directory.locking.strategy`

How to lock on the directory.

Expects a `LockingStrategyName` value, or a String representation of such value.

Defaults are specific to each directory type.

▼ *Variants of this configuration property*

- `hibernate.search.backend.indexes.<index-name>.directory.locking.strategy`
- `hibernate.search.backends.<backend-name>.directory.locking.strategy`
- `hibernate.search.backends.<backend-name>.indexes.<index-name>.directory.locking.strategy`

`hibernate.search.backend.directory.root`

The filesystem root for the directory.

Only available for the "local-filesystem" directory type.

Expects a String representing a path to an existing directory accessible in read and write mode, such as "local-filesystem".

The actual index files will be created in directory `<root>/<index-name>`.

Defaults to the JVM's working directory.

Default value: `"."`

▼ *Variants of this configuration property*

- `hibernate.search.backend.indexes.<index-name>.directory.root`
- `hibernate.search.backends.<backend-name>.directory.root`
- `hibernate.search.backends.<backend-name>.indexes.<index-name>.directory.root`

`hibernate.search.backend.directory.type`

The type of directory to use when reading from or writing to the index.

Expects a String, such as "local-filesystem". See the reference documentation for a list of available values.

Defaults to `LuceneIndexSettings.Defaults.DIRECTORY_TYPE`.

Default value: `"local-filesystem"`

▼ *Variants of this configuration property*

- `hibernate.search.backend.indexes.<index-name>.directory.type`
- `hibernate.search.backends.<backend-name>.directory.type`
- `hibernate.search.backends.<backend-name>.indexes.<index-name>.directory.type`

`hibernate.search.backend.indexing.queue_count`

The number of indexing queues assigned to each index (or each shard of each index, when sharding is enabled).

Expects a strictly positive integer value, or a string that can be parsed into an integer value.

See the reference documentation, section "Lucene backend - Indexing", for more information about this setting and its implications.

Defaults to `LuceneIndexSettings.Defaults.INDEXING_QUEUE_COUNT`.

Default value: `10`

▼ *Variants of this configuration property*

- `hibernate.search.backend.indexes.<index-name>.indexing.queue_count`
- `hibernate.search.backends.<backend-name>.indexing.queue_count`
- `hibernate.search.backends.<backend-name>.indexes.<index-name>.indexing.queue_count`

`hibernate.search.backend.indexing.queue_size`

The size of indexing queues.

Expects a strictly positive integer value, or a string that can be parsed into an integer value.

See the reference documentation, section "Lucene backend - Indexing", for more information about this setting and its implications.

Defaults to `LuceneIndexSettings.Defaults.INDEXING_QUEUE_SIZE`.

Default value: `1000`

▼ *Variants of this configuration property*

- `hibernate.search.backend.indexes.<index-name>.indexing.queue_size`
- `hibernate.search.backends.<backend-name>.indexing.queue_size`
- `hibernate.search.backends.<backend-name>.indexes.<index-name>.indexing.queue_size`

`hibernate.search.backend.io.commit_interval`

How much time may pass after an index change until the change is committed.

Only available for the "near-real-time" I/O strategy.

This effectively defines how long changes may be in an "unsafe" state, where a crash or power loss will result in data loss. For example:

- if set to 0, changes are safe as soon as the background process finishes treating a batch of changes.
- if set to 1000, changes may not be safe for an additional 1 second after the background process finishes treating a batch. There is a benefit, though: index changes occurring less than 1 second after another change may execute faster.

Note that individual write operations may trigger a forced commit (for example with the `write-sync` and `sync` indexing plan synchronization strategies in the ORM mapper), in which case you will only benefit from a non-zero commit interval during intensive indexing (mass indexer, …).

Note that committing is **not** necessary to make changes visible to search queries: the two concepts are unrelated. See `IO_REFRESH_INTERVAL`.

Expects a positive Integer value in milliseconds, such as `1000`, or a String that can be parsed into such Integer value.

Defaults to `LuceneIndexSettings.Defaults.IO_COMMIT_INTERVAL`.

Default value: `1000`

▼ *Variants of this configuration property*

- `hibernate.search.backend.indexes.<index-name>.io.commit_interval`
- `hibernate.search.backends.<backend-name>.io.commit_interval`
- `hibernate.search.backends.<backend-name>.indexes.<index-name>.io.commit_interval`

`hibernate.search.backend.io.merge.calibrate_by_deletes`

The value to pass to `LogMergePolicy.setCalibrateSizeByDeletes(boolean)`.

Expects a Boolean value such as `true` or `false`, or a String that can be parsed into such Boolean value.

The default for this setting is defined by Lucene.

▼ *Variants of this configuration property*

- `hibernate.search.backend.indexes.<index-name>.io.merge.calibrate_by_deletes`
- `hibernate.search.backends.<backend-name>.io.merge.calibrate_by_deletes`
- `hibernate.search.backends.<backend-name>.indexes.<index-name>.io.merge.calibrate_by_deletes`

`hibernate.search.backend.io.merge.factor`

The value to pass to `LogMergePolicy.setMergeFactor(int)`.

Expects a positive Integer value, or a String that can be parsed into such Integer value.

The default for this setting is defined by Lucene.

▼ *Variants of this configuration property*

- `hibernate.search.backend.indexes.<index-name>.io.merge.factor`
- `hibernate.search.backends.<backend-name>.io.merge.factor`
- `hibernate.search.backends.<backend-name>.indexes.<index-name>.io.merge.factor`

`hibernate.search.backend.io.merge.max_docs`

The value to pass to `LogMergePolicy.setMaxMergeDocs(int)`.

Expects a positive Integer value, or a String that can be parsed into such Integer value.

The default for this setting is defined by Lucene.

▼ *Variants of this configuration property*

- `hibernate.search.backend.indexes.<index-name>.io.merge.max_docs`

- `hibernate.search.backends.<backend-name>.io.merge.max_docs`
- `hibernate.search.backends.<backend-name>.indexes.<index-name>.io.merge.max_docs`

`hibernate.search.backend.io.merge.max_forced_size`

The value to pass to `LogByteSizeMergePolicy.setMaxMergeMBForForcedMerge(double)`.

Expects a positive Integer value in megabytes, or a String that can be parsed into such Integer value.

The default for this setting is defined by Lucene.

▼ *Variants of this configuration property*

- `hibernate.search.backend.indexes.<index-name>.io.merge.max_forced_size`
- `hibernate.search.backends.<backend-name>.io.merge.max_forced_size`
- `hibernate.search.backends.<backend-name>.indexes.<index-name>.io.merge.max_forced_size`

`hibernate.search.backend.io.merge.max_size`

The value to pass to `LogByteSizeMergePolicy.setMaxMergeMB(double)`.

Expects a positive Integer value in megabytes, or a String that can be parsed into such Integer value.

The default for this setting is defined by Lucene.

▼ *Variants of this configuration property*

- `hibernate.search.backend.indexes.<index-name>.io.merge.max_size`
- `hibernate.search.backends.<backend-name>.io.merge.max_size`
- `hibernate.search.backends.<backend-name>.indexes.<index-name>.io.merge.max_size`

`hibernate.search.backend.io.merge.min_size`

The value to pass to `LogByteSizeMergePolicy.setMinMergeMB(double)`.

Expects a positive Integer value in megabytes, or a String that can be parsed into such Integer value.

The default for this setting is defined by Lucene.

▼ *Variants of this configuration property*

- `hibernate.search.backend.indexes.<index-name>.io.merge.min_size`
- `hibernate.search.backends.<backend-name>.io.merge.min_size`
- `hibernate.search.backends.<backend-name>.indexes.<index-name>.io.merge.min_size`

`hibernate.search.backend.io.refresh_interval`

How much time may pass after an index write until the index reader is considered stale and re-created.

Only available for the "near-real-time" I/O strategy.

This effectively defines how out-of-date search query results may be. For example:

- If set to 0, search results will always be completely in sync with the index writes.
- If set to 1000, search results may reflect the state of the index at most 1 second ago. There is a benefit, though: in situations where the index is being frequently written to, search queries executed less than 1 second after another query may execute faster.

Note that individual write operations may trigger a forced refresh (for example with the `read-sync` and `sync` indexing plan synchronization strategies in the ORM mapper), in which case you will only benefit from a non-zero refresh interval during intensive indexing (mass indexer, …).

Expects a positive Integer value in milliseconds, such as `1000`, or a String that can be parsed into such Integer value.

Defaults to `LuceneIndexSettings.Defaults.IO_REFRESH_INTERVAL`.

Default value: `0`

▼ *Variants of this configuration property*

- `hibernate.search.backend.indexes.<index-name>.io.refresh_interval`
- `hibernate.search.backends.<backend-name>.io.refresh_interval`
- `hibernate.search.backends.<backend-name>.indexes.<index-name>.io.refresh_interval`

`hibernate.search.backend.io.strategy`

How to handle input/output, i.e. how to write to and read from indexes.

Expects a `IOStrategyName` value, or a String representation of such value.

Defaults to `LuceneIndexSettings.Defaults.IO_STRATEGY`.

▼ *Variants of this configuration property*

- `hibernate.search.backend.indexes.<index-name>.io.strategy`
- `hibernate.search.backends.<backend-name>.io.strategy`
- `hibernate.search.backends.<backend-name>.indexes.<index-name>.io.strategy`

`hibernate.search.backend.io.writer.infostream`

Whether to log the `IndexWriterConfig.setInfoStream(InfoStream)` (at the trace level) or not.

Logs are appended to the logger "org.hibernate.search.lucene.infostream".

Expects a Boolean value such as `true` or `false`, or a String that can be parsed into such Boolean value.

Default is `false`.

▼ *Variants of this configuration property*

- `hibernate.search.backend.indexes.<index-name>.io.writer.infostream`
- `hibernate.search.backends.<backend-name>.io.writer.infostream`
- `hibernate.search.backends.<backend-name>.indexes.<index-name>.io.writer.infostream`

`hibernate.search.backend.io.writer.max_buffered_docs`

The value to pass to `IndexWriterConfig.setMaxBufferedDocs(int)`.

Expects a positive Integer value, or a String that can be parsed into such Integer value.

The default for this setting is defined by Lucene.

▼ *Variants of this configuration property*

- `hibernate.search.backend.indexes.<index-name>.io.writer.max_buffered_docs`
- `hibernate.search.backends.<backend-name>.io.writer.max_buffered_docs`
- `hibernate.search.backends.<backend-name>.indexes.<index-name>.io.writer.max_buffered_docs`

`hibernate.search.backend.io.writer.ram_buffer_size`

The value to pass to `IndexWriterConfig.setRAMBufferSizeMB(double)`.

Expects a positive Integer value in megabytes, or a String that can be parsed into such Integer value.

The default for this setting is defined by Lucene.

▼ *Variants of this configuration property*

- `hibernate.search.backend.indexes.<index-name>.io.writer.ram_buffer_size`
- `hibernate.search.backends.<backend-name>.io.writer.ram_buffer_size`
- `hibernate.search.backends.<backend-name>.indexes.<index-name>.io.writer.ram_buffer_size`

`hibernate.search.backend.lucene_version`

The Lucene version to passed to analyzers when they are created.

This should be set in order to get consistent behavior when Lucene is upgraded.

Expects a Lucene `Version` object, or a String accepted by `Version.parseLeniently(java.lang.String)`

Defaults to `LuceneBackendSettings.Defaults.LUCENE_VERSION`, which may change when Hibernate Search or Lucene is upgraded, and therefore does not offer any backwards-compatibility guarantees. The recommended approach is to set this property explicitly to the version of Lucene you want to target.

▼ *Variants of this configuration property*

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

`hibernate.search.backend.multi_tenancy.strategy`

How to implement multi-tenancy.

Expects a `MultiTenancyStrategyName` value, or a String representation of such value.

Defaults to `LuceneBackendSettings.Defaults.MULTI_TENANCY_STRATEGY`.

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.multi_tenancy.strategy`

`hibernate.search.backend.query.caching.configurer`

The configurer for query caching.

Expects a single-valued or multi-valued reference to beans of type `QueryCachingConfigurer`.

Defaults to no value.

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.query.caching.configurer`

`hibernate.search.backend.sharding.number_of_shards`

The number of shards to create for the index, i.e. the number of "physical" indexes, each holding a part of the index data.

Only available for the `hash sharding strategy`.

Expects a strictly positive Integer value, such as 4, or a String that can be parsed into such Integer value.

No default: this property must be set when using the `hash` sharding strategy.

▼ *Variants of this configuration property*

- `hibernate.search.backend.indexes.<index-name>.sharding.number_of_shards`
- `hibernate.search.backends.<backend-name>.sharding.number_of_shards`
- `hibernate.search.backends.<backend-name>.indexes.<index-name>.sharding.number_of_shards`

`hibernate.search.backend.sharding.shard_identifiers`

The list of shard identifiers to accept for the index.

Only available for the `explicit sharding strategy`.

Expects either a String containing multiple shard identifiers separated by commas (','), or a `Collection<String>` containing such shard identifiers.

No default: this property must be set when using the `explicit` sharding strategy.

▼ *Variants of this configuration property*

- `hibernate.search.backend.indexes.<index-name>.sharding.shard_identifiers`
- `hibernate.search.backends.<backend-name>.sharding.shard_identifiers`
- `hibernate.search.backends.<backend-name>.indexes.<index-name>.sharding.shard_identifiers`

`hibernate.search.backend.sharding.strategy`

The sharding strategy, deciding the number of shards, their identifiers, and how to translate a routing key into a shard identifier.

Expects a String, such as "hash". See the reference documentation for a list of available values.

Defaults to `LuceneIndexSettings.Defaults.SHARDING_STRATEGY` (no sharding).

Default value: `"none"`

▼ *Variants of this configuration property*

- `hibernate.search.backend.indexes.<index-name>.sharding.strategy`
- `hibernate.search.backends.<backend-name>.sharding.strategy`
- `hibernate.search.backends.<backend-name>.indexes.<index-name>.sharding.strategy`

`hibernate.search.backend.thread_pool.size`

The size of the thread pool assigned to the backend.

Expects a strictly positive integer value, or a string that can be parsed into an integer value.

See the reference documentation, section "Lucene backend - Threads", for more information about this setting and its implications.

Defaults to the number of processor cores available to the JVM on startup.

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.thread_pool.size`

A.3. Hibernate Search Backend - Elasticsearch

`hibernate.search.backend.analysis.configurer`

The analysis configurer applied to this index.

Expects a single-valued or multi-valued reference to beans of type `ElasticsearchAnalysisConfigurer`.

Defaults to no value.

▼ *Variants of this configuration property*

- `hibernate.search.backend.indexes.<index-name>.analysis.configurer`
- `hibernate.search.backends.<backend-name>.analysis.configurer`
- `hibernate.search.backends.<backend-name>.indexes.<index-name>.analysis.configurer`

`hibernate.search.backend.client.configurer`

Deprecated, for removal: This API element is subject to removal in a future version.

Use client specific configurers instead. The setting itself is not deprecated, it just moved to the client-specific configuration.

A `ElasticsearchHttpClientConfigurer` that defines custom HTTP client configuration.

It can be used for example to tune the SSL context to accept self-signed certificates. It allows overriding other HTTP client settings, such as `USERNAME` or `MAX_CONNECTIONS_PER_ROUTE`.

Expects a reference to a bean of type `ElasticsearchHttpClientConfigurer`.

Defaults to no value.

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.client.configurer`

`hibernate.search.backend.client_factory`

Specifies which client factory to use when creating a REST client for the backend to communicate with the Elasticsearch cluster.

One of the following values can be provided:

- `jdk-rest`, no additional requirements.
- `elasticsearch-rest`, current default, requires that the `org.hibernate.search:hibernate-search-backend-elasticsearch-client-rest4` remain on the classpath.
- `elasticsearch-rest5`, requires that the `org.hibernate.search:hibernate-search-backend-elasticsearch-client-rest5` is available on the classpath.
- `opensearch-rest`, requires that the `org.hibernate.search:hibernate-search-backend-elasticsearch-client-opensearch-rest` is available on the classpath.

  Defaults to `ElasticsearchBackendSettings.Defaults.CLIENT_FACTORY`.

Default value: `"elasticsearch-rest4"`

+ .Variants of this configuration property

▼ *Details*

- `hibernate.search.backends.<backend-name>.client_factory`

`hibernate.search.backend.connection_timeout`

Deprecated, for removal: This API element is subject to removal in a future version.

Use `ClientRest4ElasticsearchBackendClientSettings.CONNECTION_TIMEOUT` instead. The setting itself is not deprecated, it just moved to the client-specific configuration.

The timeout when establishing a connection to an Elasticsearch server.

Expects a positive Integer value in milliseconds, such as `3000`, or a String that can be parsed into such Integer value.

Defaults to `ElasticsearchBackendSettings.Defaults.CONNECTION_TIMEOUT`.

Default value: `1000`

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.connection_timeout`

`hibernate.search.backend.discovery.enabled`

Deprecated, for removal: This API element is subject to removal in a future version.

Use `ClientRest4ElasticsearchBackendClientSettings.DISCOVERY_ENABLED` instead. The setting itself is not deprecated, it just moved to the client-specific configuration.

Whether automatic discovery of nodes in the Elasticsearch cluster is enabled.

Expects a Boolean value such as `true` or `false`, or a string that can be parsed into a Boolean value.

Defaults to `ElasticsearchBackendSettings.Defaults.DISCOVERY_ENABLED`.

Default value: `false`

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.discovery.enabled`

`hibernate.search.backend.discovery.refresh_interval`

Deprecated, for removal: This API element is subject to removal in a future version.

Use `ClientRest4ElasticsearchBackendClientSettings.DISCOVERY_REFRESH_INTERVAL` instead. The setting itself is not deprecated, it just moved to the client-specific configuration.

The time interval between two executions of the automatic discovery, if enabled.

Expects a positive Integer value in seconds, such as `2`, or a String that can be parsed into such Integer value.

Defaults to `ElasticsearchBackendSettings.Defaults.DISCOVERY_REFRESH_INTERVAL`.

Default value: `10`

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.discovery.refresh_interval`

`hibernate.search.backend.dynamic_mapping`

The default for the `dynamic_mapping` attribute in the Elasticsearch mapping.

In case of dynamic fields with field templates, this setting will be ignored, since field templates force `dynamic_mapping` to `DynamicMapping.TRUE`.

Defaults to `ElasticsearchIndexSettings.Defaults.DYNAMIC_MAPPING`.

▼ *Variants of this configuration property*

- `hibernate.search.backend.indexes.<index-name>.dynamic_mapping`
- `hibernate.search.backends.<backend-name>.dynamic_mapping`
- `hibernate.search.backends.<backend-name>.indexes.<index-name>.dynamic_mapping`

`hibernate.search.backend.hosts`

The hostname and ports of the Elasticsearch servers to connect to.

Expects a String representing a hostname and port such as `localhost` or `es.mycompany.com:4400`, or a String containing multiple such hostname-and-port strings separated by commas, or a `Collection<String>` containing such hostname-and-port strings.

Multiple servers may be specified for load-balancing: requests will be assigned to each host in turns.

Setting this property at the same time as `URIS` will lead to an exception being thrown on startup.

Defaults to `ElasticsearchBackendSettings.Defaults.HOSTS`.

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.hosts`

`hibernate.search.backend.indexing.max_bulk_size`

The maximum size of bulk requests created when processing indexing queues.

Expects a strictly positive integer value, or a string that can be parsed into an integer value.

See the reference documentation, section "Elasticsearch backend - Indexing", for more information about this setting and its implications.

Defaults to `ElasticsearchIndexSettings.Defaults.INDEXING_MAX_BULK_SIZE`.

Default value: `100`

▼ *Variants of this configuration property*

- `hibernate.search.backend.indexes.<index-name>.indexing.max_bulk_size`
- `hibernate.search.backends.<backend-name>.indexing.max_bulk_size`
- `hibernate.search.backends.<backend-name>.indexes.<index-name>.indexing.max_bulk_size`

`hibernate.search.backend.indexing.queue_count`

The number of indexing queues assigned to each index.

Expects a strictly positive integer value, or a string that can be parsed into an integer value.

Defaults to `ElasticsearchIndexSettings.Defaults.INDEXING_QUEUE_COUNT`.

See the reference documentation, section "Elasticsearch backend - Indexing", for more information about this setting and its implications.

Default value: `10`

▼ *Variants of this configuration property*

- `hibernate.search.backend.indexes.<index-name>.indexing.queue_count`
- `hibernate.search.backends.<backend-name>.indexing.queue_count`
- `hibernate.search.backends.<backend-name>.indexes.<index-name>.indexing.queue_count`

`hibernate.search.backend.indexing.queue_size`

The size of indexing queues.

Expects a strictly positive integer value, or a string that can be parsed into an integer value.

See the reference documentation, section "Elasticsearch backend - Indexing", for more information about this setting and its implications.

Defaults to `ElasticsearchIndexSettings.Defaults.INDEXING_QUEUE_SIZE`.

Default value: `1000`

▼ *Variants of this configuration property*

- `hibernate.search.backend.indexes.<index-name>.indexing.queue_size`
- `hibernate.search.backends.<backend-name>.indexing.queue_size`
- `hibernate.search.backends.<backend-name>.indexes.<index-name>.indexing.queue_size`

`hibernate.search.backend.layout.strategy`

How to determine index names and aliases.

Expects a reference to a bean of type `IndexLayoutStrategy`.

Defaults to the `simple` strategy:

- The non-alias name follows the format `<hibernateSearchIndexName>-<6 digits>`
- The write alias follows the format `<hibernateSearchIndexName>-write`
- The read alias follows the format `<hibernateSearchIndexName>-read`

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.layout.strategy`

`hibernate.search.backend.log.json_pretty_printing`

Whether JSON included in logs should be pretty-printed (indented, with line breaks).

Expects a Boolean value such as `true` or `false`, or a string that can be parsed into a Boolean value.

Defaults to `ElasticsearchBackendSettings.Defaults.LOG_JSON_PRETTY_PRINTING`.

Default value: `false`

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.log.json_pretty_printing`

`hibernate.search.backend.mapping.type_name.strategy`

How to map documents to their type name, i.e. how to determine the type name of a document in search hits.

Expects a `TypeNameMappingStrategyName` value, or a String representation of such value.

Defaults to `ElasticsearchBackendSettings.Defaults.MAPPING_TYPE_NAME_STRATEGY`.

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.mapping.type_name.strategy`

`hibernate.search.backend.max_connections`

Deprecated, for removal: This API element is subject to removal in a future version.

Use `ClientRest4ElasticsearchBackendClientSettings.MAX_CONNECTIONS` instead. The setting itself is not deprecated, it just moved to the client-specific configuration.

The maximum number of simultaneous connections to the Elasticsearch cluster, all hosts taken together.

Expects a positive Integer value, such as `40`, or a String that can be parsed into such Integer value.

Defaults to `ElasticsearchBackendSettings.Defaults.MAX_CONNECTIONS`.

Default value: `40`

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.max_connections`

`hibernate.search.backend.max_connections_per_route`

Deprecated, for removal: This API element is subject to removal in a future version.

Use `ClientRest4ElasticsearchBackendClientSettings.MAX_CONNECTIONS_PER_ROUTE` instead. The setting itself is not deprecated, it just moved to the client-specific configuration.

The maximum number of simultaneous connections to each host of the Elasticsearch cluster.

Expects a positive Integer value, such as `20`, or a String that can be parsed into such Integer value.

Defaults to `ElasticsearchBackendSettings.Defaults.MAX_CONNECTIONS_PER_ROUTE`.

Default value: `20`

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.max_connections_per_route`

`hibernate.search.backend.max_keep_alive`

Deprecated, for removal: This API element is subject to removal in a future version.

Use `ClientRest4ElasticsearchBackendClientSettings.MAX_KEEP_ALIVE` instead. The setting itself is not deprecated, it just moved to the client-specific configuration.

How long connections to the Elasticsearch cluster can be kept idle.

Expects a positive Long value of milliseconds, such as 60000, or a String that can be parsed into such Integer value.

If the response from an Elasticsearch cluster contains a `Keep-Alive` header, then the effective max idle time will be whichever is lower: the duration from the `Keep-Alive` header or the value of this property (if set).

If this property is not set, only the `Keep-Alive` header is considered, and if it's absent, idle connections will be kept forever.

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.max_keep_alive`

`hibernate.search.backend.multi_tenancy.strategy`

How to implement multi-tenancy.

Expects a `MultiTenancyStrategyName` value, or a String representation of such value.

Defaults to `ElasticsearchBackendSettings.Defaults.MULTI_TENANCY_STRATEGY`.

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.multi_tenancy.strategy`

`hibernate.search.backend.password`

The password to send when connecting to the Elasticsearch servers (HTTP authentication).

Expects a String.

Defaults to no username (anonymous access).

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.password`

`hibernate.search.backend.path_prefix`

Property for specifying the path prefix prepended to the request end point. Use the path prefix if your Elasticsearch instance is located at a specific context path.

Defaults to `ElasticsearchBackendSettings.Defaults.PATH_PREFIX`.

Default value: `""`

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.path_prefix`

`hibernate.search.backend.protocol`

The protocol to use when connecting to the Elasticsearch servers.

Expects a String: either `http` or `https`.

Setting this property at the same time as `URIS` will lead to an exception being thrown on startup.

Defaults to `ElasticsearchBackendSettings.Defaults.PROTOCOL`.

Default value: `"http"`

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.protocol`

`hibernate.search.backend.query.shard_failure.ignore`

This property defines if partial shard failures are ignored.

In case all shards fail, Elasticsearch cluster will return a 400 status code itself, but if only some of the shards fail, then the client will

receive a successful partial response from the shards that were successful.

To prevent getting any partial results this setting can be set to `false`. While if the partial failures should be ignored and considered as valid results then the value should be set to `true`.

Expects a Boolean value such as `true` or `false`, or a string that can be parsed into a Boolean value.

Defaults to `ElasticsearchBackendSettings.Defaults.QUERY_SHARD_FAILURE_IGNORE`.

Default value: `false`

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.query.shard_failure.ignore`

`hibernate.search.backend.read_timeout`

Deprecated, for removal: This API element is subject to removal in a future version.

Use `ClientRest4ElasticsearchBackendClientSettings.READ_TIMEOUT` instead. The setting itself is not deprecated, it just moved to the client-specific configuration.

The timeout when reading responses from an Elasticsearch server.

Expects a positive Integer value in milliseconds, such as `60000`, or a String that can be parsed into such Integer value.

Defaults to `ElasticsearchBackendSettings.Defaults.READ_TIMEOUT`.

Default value: `30000`

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.read_timeout`

`hibernate.search.backend.request_timeout`

Deprecated, for removal: This API element is subject to removal in a future version.

Use `ClientRest4ElasticsearchBackendClientSettings.REQUEST_TIMEOUT` instead. The setting itself is not deprecated, it just moved to the client-specific configuration.

The timeout when executing a request to an Elasticsearch server.

This includes the time needed to establish a connection, send the request and read the response.

Expects a positive Integer value in milliseconds, such as 60000, or a String that can be parsed into such Integer value.

Defaults to no request timeout.

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.request_timeout`

`hibernate.search.backend.schema_management.mapping_file`

The path to a mappings file, allowing custom mappings for indexes created by Hibernate Search as part of schema management.

Expects a string representing the path to a UTF-8-encoded file in the classpath. The file must contain index settings expressed in JSON format, with the exact same syntax as expected by the Elasticsearch server under the "mappings" property when defining the mapping for an index.

The file does not need to contain the full mapping: Hibernate Search will automatically inject missing properties (index fields) in the given mapping.

Conflicts between the given mapping and the mapping generated by Hibernate Search will be handled as follows:

- Mapping parameters other than `properties` at the mapping root will be those from the given mapping; those generated by Hibernate Search will be ignored.
- `properties` will be merged, using properties defined in both the given mapping and the mapping generated by Hibernate Search. If a property is defined on both sides, mapping parameters from the given mapping will be used, except for `properties`, which will be merged recursively in the same way.

Defaults to no value, meaning only index mappings generated by Hibernate Search will be used.

▼ *Variants of this configuration property*

- `hibernate.search.backend.indexes.<index-name>.schema_management.mapping_file`
- `hibernate.search.backends.<backend-name>.schema_management.mapping_file`
- `hibernate.search.backends.<backend-name>.indexes.<index-name>.schema_management.mapping_file`

`hibernate.search.backend.schema_management.minimal_required_status`

The minimal required status of an index on startup, before Hibernate Search can start using it.

Expects an `IndexStatus` value, or a String representation of such value.

Defaults to `yellow` when targeting an Elasticsearch distribution that supports index status checking, and to no value (no requirement) when targeting an Elasticsearch distribution that does not support index status checking (like Amazon OpenSearch Serverless).

▼ *Variants of this configuration property*

- `hibernate.search.backend.indexes.<index-name>.schema_management.minimal_required_status`
- `hibernate.search.backends.<backend-name>.schema_management.minimal_required_status`
- `hibernate.search.backends.<backend-name>.indexes.<index-name>.schema_management.minimal_required_status`

`hibernate.search.backend.schema_management.minimal_required_status_wait_timeout`

The timeout when waiting for the `required status`.

Expects a positive Integer value in milliseconds, such as `60000`, or a String that can be parsed into such Integer value.

Defaults to `ElasticsearchIndexSettings.Defaults.SCHEMA_MANAGEMENT_MINIMAL_REQUIRED_STATUS_WAIT_TIMEOUT`.

Default value: `10000`

▼ *Variants of this configuration property*

- `hibernate.search.backend.indexes.<index-name>.schema_management.minimal_required_status_wait_timeout`
- `hibernate.search.backends.<backend-name>.schema_management.minimal_required_status_wait_timeout`
- `hibernate.search.backends.<backend-name>.indexes.<index-name>.schema_management.minimal_required_status_wait_timeout`

`hibernate.search.backend.schema_management.settings_file`

The path to a settings file, allowing custom settings for indexes created by Hibernate Search as part of schema management.

Expects a string representing the path to a UTF-8-encoded file in the classpath. The file must contain index settings expressed in JSON format, with the exact same syntax as expected by the Elasticsearch server under the "settings" property when creating an index. For example, if the file content is `{"index.codec": "best_compression"}`, it will set `index.codec` to `best_compression`.

Note that the settings generated by Hibernate Search will be overridden in case of conflict of some definitions. For instance, if an analyzer "myAnalyzer" is defined by the `ANALYSIS_CONFIGURER` **and** this settings file, the definition from the settings file will take precedence. If it is only defined in either the analysis configurer or the settings file, but not both, it will be preserved as-is.

Defaults to no value, meaning only index settings generated by Hibernate Search will be used.

▼ *Variants of this configuration property*

- `hibernate.search.backend.indexes.<index-name>.schema_management.settings_file`
- `hibernate.search.backends.<backend-name>.schema_management.settings_file`
- `hibernate.search.backends.<backend-name>.indexes.<index-name>.schema_management.settings_file`

`hibernate.search.backend.scroll_timeout`

Property for specifying the maximum duration a `scroll` will be usable if no other results are fetched from Elasticsearch.

Expects a positive Integer value in seconds, such as 60, or a String that can be parsed into such Integer value.

Defaults to `ElasticsearchBackendSettings.Defaults.SCROLL_TIMEOUT`.

Default value: `60`

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.scroll_timeout`

`hibernate.search.backend.thread_pool.size`

The size of the thread pool assigned to the backend.

Expects a strictly positive integer value, or a string that can be parsed into an integer value.

See the reference documentation, section "Elasticsearch backend - Threads", for more information about this setting and its implications.

Defaults to the number of processor cores available to the JVM on startup.

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.thread_pool.size`

`hibernate.search.backend.uris`

The protocol, hostname and ports of the Elasticsearch servers to connect to.

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

All the URIs must specify the same protocol.

Setting this property at the same time as `HOSTS` or `PROTOCOL` will lead to an exception being thrown on startup.

Defaults to `http://localhost:9200`, unless `HOSTS` or `PROTOCOL` are set, in which case they take precedence.

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.uris`

`hibernate.search.backend.username`

The username to send when connecting to the Elasticsearch servers (HTTP authentication).

Expects a String.

Defaults to no username (anonymous access).

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.username`

`hibernate.search.backend.version`

The version of Elasticsearch running on the Elasticsearch cluster.

Expects either an `ElasticsearchVersion` object, or a String that can be `parsed` in such an object.

No default: if not provided, the version will be resolved automatically by sending a request to the Elasticsearch cluster on startup.

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.version`

`hibernate.search.backend.version_check.enabled`

Whether check version of the Elasticsearch cluster is enabled.

Expects a Boolean value such as `true` or `false`, or a string that can be parsed into a Boolean value.

Defaults to `true` when the `VERSION` is unconfigured or set to a distribution that supports version checking, and to `false` when the `VERSION` is set to a distribution that does not support version checking (like Amazon OpenSearch Serverless).

Default value: `true`

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.version_check.enabled`

A.4. Hibernate Search Backend - Elasticsearch - AWS integration

`hibernate.search.backend.aws.credentials.access_key_id`

The AWS access key ID when using `static credentials`.

Expects a String value such as `AKIDEXAMPLE`.

No default: must be provided when signing is enabled and the credentials type is set to `ElasticsearchAwsCredentialsTypeNames.STATIC`.

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.aws.credentials.access_key_id`

`hibernate.search.backend.aws.credentials.secret_access_key`

The AWS secret access key when using `static credentials`.

Expects a String value such as `wJalrXUtnFEMI/K7MDENG+bPxRfiCYEXAMPLEKEY`

No default: must be provided when signing is enabled and the credentials type is set to `ElasticsearchAwsCredentialsTypeNames.STATIC`.

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.aws.credentials.secret_access_key`

`hibernate.search.backend.aws.credentials.type`

The type of credentials to use when signing is `enabled`.

Expects one of the names listed as constants in `ElasticsearchAwsCredentialsTypeNames`.

Defaults to `ElasticsearchAwsBackendSettings.Defaults.CREDENTIALS_TYPE`.

Default value: `"default"`

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.aws.credentials.type`

`hibernate.search.backend.aws.region`

The AWS region.

Expects a String value such as `us-east-1`.

No default: must be provided when signing is enabled.

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.aws.region`

`hibernate.search.backend.aws.signing.enabled`

Whether requests should be signed using the AWS credentials.

Expects a Boolean value such as `true` or `false`, or a string that can be parsed into a Boolean value.

Defaults to `ElasticsearchAwsBackendSettings.Defaults.SIGNING_ENABLED`.

Default value: `false`

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.aws.signing.enabled`

A.5. Hibernate Search Backend - Elasticsearch client based on the low-level Elasticsearch REST client

`hibernate.search.backend.client.configurer`

A `ElasticsearchHttpClientConfigurer` that defines custom HTTP client configuration.

It can be used for example to tune the SSL context to accept self-signed certificates. It allows overriding other HTTP client settings, such as `USERNAME` or `MAX_CONNECTIONS_PER_ROUTE`.

Expects a reference to a bean of type `ElasticsearchHttpClientConfigurer`.

Defaults to no value.

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.client.configurer`

`hibernate.search.backend.connection_timeout`

The timeout when establishing a connection to an Elasticsearch server.

Expects a positive Integer value in milliseconds, such as `3000`, or a String that can be parsed into such Integer value.

Defaults to `ClientRest4ElasticsearchBackendClientSettings.Defaults.CONNECTION_TIMEOUT`.

Default value: `1000`

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.connection_timeout`

`hibernate.search.backend.discovery.enabled`

Whether automatic discovery of nodes in the Elasticsearch cluster is enabled.

Expects a Boolean value such as `true` or `false`, or a string that can be parsed into a Boolean value.

Defaults to `ClientRest4ElasticsearchBackendClientSettings.Defaults.DISCOVERY_ENABLED`.

Default value: `false`

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.discovery.enabled`

`hibernate.search.backend.discovery.refresh_interval`

The time interval between two executions of the automatic discovery, if enabled.

Expects a positive Integer value in seconds, such as `2`, or a String that can be parsed into such Integer value.

Defaults to `ClientRest4ElasticsearchBackendClientSettings.Defaults.DISCOVERY_REFRESH_INTERVAL`.

Default value: `10`

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.discovery.refresh_interval`

`hibernate.search.backend.max_connections`

The maximum number of simultaneous connections to the Elasticsearch cluster, all hosts taken together.

Expects a positive Integer value, such as `40`, or a String that can be parsed into such Integer value.

Defaults to `ClientRest4ElasticsearchBackendClientSettings.Defaults.MAX_CONNECTIONS`.

Default value: `40`

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.max_connections`

`hibernate.search.backend.max_connections_per_route`

The maximum number of simultaneous connections to each host of the Elasticsearch cluster.

Expects a positive Integer value, such as `20`, or a String that can be parsed into such Integer value.

Defaults to `ClientRest4ElasticsearchBackendClientSettings.Defaults.MAX_CONNECTIONS_PER_ROUTE`.

Default value: `20`

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.max_connections_per_route`

`hibernate.search.backend.max_keep_alive`

How long connections to the Elasticsearch cluster can be kept idle.

Expects a positive Long value of milliseconds, such as 60000, or a String that can be parsed into such Integer value.

If the response from an Elasticsearch cluster contains a `Keep-Alive` header, then the effective max idle time will be whichever is lower: the duration from the `Keep-Alive` header or the value of this property (if set).

If this property is not set, only the `Keep-Alive` header is considered, and if it's absent, idle connections will be kept forever.

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.max_keep_alive`

`hibernate.search.backend.read_timeout`

The timeout when reading responses from an Elasticsearch server.

Expects a positive Integer value in milliseconds, such as `60000`, or a String that can be parsed into such Integer value.

Defaults to `ClientRest4ElasticsearchBackendClientSettings.Defaults.READ_TIMEOUT`.

Default value: `30000`

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.read_timeout`

`hibernate.search.backend.request_timeout`

The timeout when executing a request to an Elasticsearch server.

This includes the time needed to establish a connection, send the request and read the response.

Expects a positive Integer value in milliseconds, such as 60000, or a String that can be parsed into such Integer value.

Defaults to no request timeout.

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.request_timeout`

A.6. Hibernate Search Backend - Elasticsearch client based on the low-level Elasticsearch REST client

`hibernate.search.backend.client.configurer`

A `ElasticsearchHttpClientConfigurer` that defines custom HTTP client configuration.

It can be used for example to tune the SSL context to accept self-signed certificates. It allows overriding other HTTP client settings, such as `ElasticsearchBackendSettings.USERNAME` or `MAX_CONNECTIONS_PER_ROUTE`.

Expects a reference to a bean of type `ElasticsearchHttpClientConfigurer`.

Defaults to no value.

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.client.configurer`

`hibernate.search.backend.connection_timeout`

The timeout when establishing a connection to an Elasticsearch server.

Expects a positive Integer value in milliseconds, such as `3000`, or a String that can be parsed into such Integer value.

Defaults to `ClientRest5ElasticsearchBackendClientSettings.Defaults.CONNECTION_TIMEOUT`.

Default value: `1000`

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.connection_timeout`

`hibernate.search.backend.discovery.enabled`

Whether automatic discovery of nodes in the Elasticsearch cluster is enabled.

Expects a Boolean value such as `true` or `false`, or a string that can be parsed into a Boolean value.

Defaults to `ClientRest5ElasticsearchBackendClientSettings.Defaults.DISCOVERY_ENABLED`.

Default value: `false`

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.discovery.enabled`

`hibernate.search.backend.discovery.refresh_interval`

The time interval between two executions of the automatic discovery, if enabled.

Expects a positive Integer value in seconds, such as `2`, or a String that can be parsed into such Integer value.

Defaults to `ClientRest5ElasticsearchBackendClientSettings.Defaults.DISCOVERY_REFRESH_INTERVAL`.

Default value: `10`

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.discovery.refresh_interval`

`hibernate.search.backend.max_connections`

The maximum number of simultaneous connections to the Elasticsearch cluster, all hosts taken together.

Expects a positive Integer value, such as `40`, or a String that can be parsed into such Integer value.

Defaults to `ClientRest5ElasticsearchBackendClientSettings.Defaults.MAX_CONNECTIONS`.

Default value: `40`

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.max_connections`

`hibernate.search.backend.max_connections_per_route`

The maximum number of simultaneous connections to each host of the Elasticsearch cluster.

Expects a positive Integer value, such as `20`, or a String that can be parsed into such Integer value.

Defaults to `ClientRest5ElasticsearchBackendClientSettings.Defaults.MAX_CONNECTIONS_PER_ROUTE`.

Default value: `20`

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.max_connections_per_route`

`hibernate.search.backend.max_keep_alive`

How long connections to the Elasticsearch cluster can be kept idle.

Expects a positive Long value of milliseconds, such as 60000, or a String that can be parsed into such Integer value.

If the response from an Elasticsearch cluster contains a `Keep-Alive` header, then the effective max idle time will be whichever is lower: the duration from the `Keep-Alive` header or the value of this property (if set).

If this property is not set, only the `Keep-Alive` header is considered, and if it's absent, idle connections will be kept forever.

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.max_keep_alive`

`hibernate.search.backend.read_timeout`

The timeout when reading responses from an Elasticsearch server.

Expects a positive Integer value in milliseconds, such as `60000`, or a String that can be parsed into such Integer value.

Defaults to `ClientRest5ElasticsearchBackendClientSettings.Defaults.READ_TIMEOUT`.

Default value: `30000`

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.read_timeout`

`hibernate.search.backend.request_timeout`

The timeout when executing a request to an Elasticsearch server.

This includes the time needed to establish a connection, send the request and read the response.

Expects a positive Integer value in milliseconds, such as 60000, or a String that can be parsed into such Integer value.

Defaults to no request timeout.

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.request_timeout`

A.7. Hibernate Search Backend - Elasticsearch client based on the low-level OpenSearch client

`hibernate.search.backend.client.configurer`

A `ElasticsearchHttpClientConfigurer` that defines custom HTTP client configuration.

It can be used for example to tune the SSL context to accept self-signed certificates. It allows overriding other HTTP client settings, such as `ElasticsearchBackendSettings.USERNAME` or `MAX_CONNECTIONS_PER_ROUTE`.

Expects a reference to a bean of type `ElasticsearchHttpClientConfigurer`.

Defaults to no value.

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.client.configurer`

`hibernate.search.backend.connection_timeout`

The timeout when establishing a connection to an Elasticsearch server.

Expects a positive Integer value in milliseconds, such as `3000`, or a String that can be parsed into such Integer value.

Defaults to `ClientOpenSearchElasticsearchBackendClientSettings.Defaults.CONNECTION_TIMEOUT`.

Default value: `1000`

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.connection_timeout`

`hibernate.search.backend.discovery.enabled`

Whether automatic discovery of nodes in the Elasticsearch cluster is enabled.

Expects a Boolean value such as `true` or `false`, or a string that can be parsed into a Boolean value.

Defaults to `ClientOpenSearchElasticsearchBackendClientSettings.Defaults.DISCOVERY_ENABLED`.

Default value: `false`

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.discovery.enabled`

`hibernate.search.backend.discovery.refresh_interval`

The time interval between two executions of the automatic discovery, if enabled.

Expects a positive Integer value in seconds, such as `2`, or a String that can be parsed into such Integer value.

Defaults to `ClientOpenSearchElasticsearchBackendClientSettings.Defaults.DISCOVERY_REFRESH_INTERVAL`.

Default value: `10`

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.discovery.refresh_interval`

`hibernate.search.backend.max_connections`

The maximum number of simultaneous connections to the Elasticsearch cluster, all hosts taken together.

Expects a positive Integer value, such as `40`, or a String that can be parsed into such Integer value.

Defaults to `ClientOpenSearchElasticsearchBackendClientSettings.Defaults.MAX_CONNECTIONS`.

Default value: `40`

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.max_connections`

`hibernate.search.backend.max_connections_per_route`

The maximum number of simultaneous connections to each host of the Elasticsearch cluster.

Expects a positive Integer value, such as `20`, or a String that can be parsed into such Integer value.

Defaults to `ClientOpenSearchElasticsearchBackendClientSettings.Defaults.MAX_CONNECTIONS_PER_ROUTE`.

Default value: `20`

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.max_connections_per_route`

`hibernate.search.backend.max_keep_alive`

How long connections to the Elasticsearch cluster can be kept idle.

Expects a positive Long value of milliseconds, such as 60000, or a String that can be parsed into such Integer value.

If the response from an Elasticsearch cluster contains a `Keep-Alive` header, then the effective max idle time will be whichever is lower: the duration from the `Keep-Alive` header or the value of this property (if set).

If this property is not set, only the `Keep-Alive` header is considered, and if it's absent, idle connections will be kept forever.

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.max_keep_alive`

`hibernate.search.backend.read_timeout`

The timeout when reading responses from an Elasticsearch server.

Expects a positive Integer value in milliseconds, such as `60000`, or a String that can be parsed into such Integer value.

Defaults to `ClientOpenSearchElasticsearchBackendClientSettings.Defaults.READ_TIMEOUT`.

Default value: `30000`

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.read_timeout`

`hibernate.search.backend.request_timeout`

The timeout when executing a request to an Elasticsearch server.

This includes the time needed to establish a connection, send the request and read the response.

Expects a positive Integer value in milliseconds, such as 60000, or a String that can be parsed into such Integer value.

Defaults to no request timeout.

▼ *Variants of this configuration property*

- `hibernate.search.backends.<backend-name>.request_timeout`

A.8. Hibernate Search ORM Integration

`hibernate.search.automatic_indexing.enable_dirty_check`

Deprecated.

This setting will be removed in a future version. There will be no alternative provided to replace it. After the removal of this property in a future version, a dirty check will always be performed when considering whether to trigger reindexing.

Whether to check if dirty properties are relevant to indexing before actually reindexing an entity.

When enabled, re-indexing of an entity is skipped if the only changes are on properties that are not used when indexing. This feature is considered safe and thus enabled by default.

Expects a Boolean value such as `true` or `false`, or a string that can be parsed into a Boolean value.

Defaults to `HibernateOrmMapperSettings.Defaults.AUTOMATIC_INDEXING_ENABLE_DIRTY_CHECK`.

Default value: `true`

`hibernate.search.automatic_indexing.enabled`

Deprecated.

Use `INDEXING_LISTENERS_ENABLED` instead.

Whether listener-triggered indexing is enabled, i.e. whether changes to entities in a Hibernate ORM session are detected automatically and lead to reindexing.

Expects a Boolean value such as `true` or `false`, or a string that can be parsed into a Boolean value.

Defaults to `HibernateOrmMapperSettings.Defaults.AUTOMATIC_INDEXING_ENABLED`.

Default value: `true`

`hibernate.search.automatic_indexing.strategy`

Deprecated.

Use `INDEXING_LISTENERS_ENABLED` instead (caution: it expects a boolean value).

How to enable or disable listener-triggered indexing.

Expects a `AutomaticIndexingStrategyName` value, or a String representation of such value.

Defaults to `HibernateOrmMapperSettings.Defaults.AUTOMATIC_INDEXING_STRATEGY`.

`hibernate.search.automatic_indexing.synchronization.strategy`

Deprecated.

Use `INDEXING_PLAN_SYNCHRONIZATION_STRATEGY` instead.

How to synchronize between application threads and indexing triggered by the `SearchIndexingPlan`.

Expects one of the strings defined in `AutomaticIndexingSynchronizationStrategyNames`, or a reference to a bean of type `AutomaticIndexingSynchronizationStrategy`.

Defaults to `HibernateOrmMapperSettings.Defaults.AUTOMATIC_INDEXING_SYNCHRONIZATION_STRATEGY`.

`hibernate.search.coordination.strategy`

How to coordinate between nodes of a distributed application.

Expects a reference to a coordination strategy; see the reference documentation for available strategies and the relevant Maven dependencies.

Defaults to `HibernateOrmMapperSettings.Defaults.COORDINATION_STRATEGY`.

`hibernate.search.enabled`

Whether Hibernate Search is enabled or disabled.

Expects a Boolean value such as `true` or `false`, or a string that can be parsed into a Boolean value.

Defaults to `HibernateOrmMapperSettings.Defaults.ENABLED`.

Default value: `true`

`hibernate.search.indexing.listeners.enabled`

Whether Hibernate ORM listeners that detect entity changes and automatically trigger indexing operations are enabled.

Expects a Boolean value such as `true` or `false`, or a string that can be parsed into a Boolean value.

Defaults to `HibernateOrmMapperSettings.Defaults.INDEXING_LISTENERS_ENABLED`.

Default value: `true`

`hibernate.search.indexing.mass.default_clean_operation`

The default index cleaning operation to apply during mass indexing, unless configured explicitly.

Expects a `MassIndexingDefaultCleanOperation` value, or a String representation of such value.

Defaults to `HibernateOrmMapperSettings.Defaults.INDEXING_MASS_DEFAULT_CLEAN_OPERATION`.

`hibernate.search.indexing.plan.synchronization.strategy`

How to synchronize between application threads and indexing triggered by the `SearchIndexingPlan`.

Expects one of the strings defined in `IndexingPlanSynchronizationStrategyNames`, or a reference to a bean of type `IndexingPlanSynchronizationStrategy`.

Defaults to `HibernateOrmMapperSettings.Defaults.INDEXING_PLAN_SYNCHRONIZATION_STRATEGY`.

`hibernate.search.mapping.build_missing_discovered_jandex_indexes`

When `annotation processing is enabled` (the default), whether Hibernate Search should automatically build Jandex indexes for types registered for annotation processing (entities in particular), to ensure that all "root mapping" annotations in those JARs (e.g. `ProjectionConstructor`) are taken into account.

Expects a Boolean value such as `true` or `false`, or a string that can be parsed into a Boolean value.

Defaults to `HibernateOrmMapperSettings.Defaults.MAPPING_BUILD_MISSING_DISCOVERED_JANDEX_INDEXES`.

Default value: `true`

`hibernate.search.mapping.configurer`

A configurer for the Hibernate Search mapping.

Expects a single-valued or multi-valued reference to beans of type `HibernateOrmSearchMappingConfigurer`.

Defaults to no value.

`hibernate.search.mapping.discover_annotated_types_from_root_mapping_annotations`

When `annotation processing is enabled` (the default), whether Hibernate Search should automatically discover annotated types present in the Jandex index that are also annotated with `root mapping annotations`.

When enabled, if an annotation meta-annotated with `RootMapping` is found in the Jandex index, and a type annotated with that annotation (e.g. `SearchEntity` or `ProjectionConstructor`) is found in the Jandex index, then that type will automatically be scanned for mapping annotations, even if the type wasn't explicitly added.

Expects a Boolean value such as `true` or `false`, or a string that can be parsed into a Boolean value.

Defaults to `HibernateOrmMapperSettings.Defaults.MAPPING_DISCOVER_ANNOTATED_TYPES_FROM_ROOT_MAPPING_ANNOTATIONS`.

Default value: `true`

`hibernate.search.mapping.process_annotations`

Whether annotations should be automatically processed for entity types, as well as nested types in those entity types, for instance `index-embedded` types.

Expects a Boolean value such as `true` or `false`, or a string that can be parsed into a Boolean value.

Defaults to `HibernateOrmMapperSettings.Defaults.MAPPING_PROCESS_ANNOTATIONS`.

Default value: `true`

`hibernate.search.multi_tenancy.tenant_identifier_converter`

How to convert tenant identifier to and form the string representation.

Converts a tenant identifier to a string representation to be written to the index, and converts to its object representation from a string when a new Hibernate ORM session must be opened.

When multi-tenancy is enabled, and non-string tenant identifiers are used a custom converter **must** be provided through this property.

Defaults to `HibernateOrmMapperSettings.Defaults.MULTI_TENANCY_TENANT_IDENTIFIER_CONVERTER`. This converter only supports string tenant identifiers and will fail if some other type of identifiers is used.

`hibernate.search.multi_tenancy.tenant_ids`

An exhaustive list of all tenant identifiers that can be used by the application when multi-tenancy is enabled.

Expects either a String representing multiple tenant IDs separated by commas, or a `Collection<String>` containing tenant IDs.

No default; this property may have to be set explicitly depending on the `coordination strategy`.

`hibernate.search.query.loading.cache_lookup.strategy`

How to look up entities in the second-level cache when loading entities for a search query.

Expects a `EntityLoadingCacheLookupStrategy` value, or a String representation of such value.

Defaults to `HibernateOrmMapperSettings.Defaults.QUERY_LOADING_CACHE_LOOKUP_STRATEGY`.

`hibernate.search.query.loading.fetch_size`

How many entities to load per database query when loading entities for a search query.

Expects a strictly positive Integer value, such as `100`, or a String that can be parsed into such Integer value.

Defaults to `HibernateOrmMapperSettings.Defaults.QUERY_LOADING_FETCH_SIZE`.

Default value: `100`

`hibernate.search.schema_management.strategy`

How indexes and their schema are created, updated, validated or dropped on startup and shutdown.

Expects a `SchemaManagementStrategyName` value, or a String representation of such value.

Defaults to `HibernateOrmMapperSettings.Defaults.SCHEMA_MANAGEMENT_STRATEGY`.

A.9. Hibernate Search ORM Integration - Coordination - Outbox Polling

`hibernate.search.coordination.entity.mapping.agent.catalog`

The database catalog to use for the agent table.

Only available when "hibernate.search.coordination.strategy" is "outbox-polling".

Defaults to the default catalog configured in Hibernate ORM. See `MappingSettings.DEFAULT_CATALOG`.

`hibernate.search.coordination.entity.mapping.agent.schema`

The database schema to use for the agent table.

Only available when "hibernate.search.coordination.strategy" is "outbox-polling".

Defaults to the default schema configured in Hibernate ORM. See `MappingSettings.DEFAULT_SCHEMA`.

`hibernate.search.coordination.entity.mapping.agent.table`

The name of the agent table.

Only available when "hibernate.search.coordination.strategy" is "outbox-polling".

The default for this value is "HSEARCH_AGENT".

Default value: `"HSEARCH_AGENT"`

`hibernate.search.coordination.entity.mapping.agent.uuid_gen_strategy`

The name of UUID generator strategy to be used by `UuidGenerator` for the agent table.

Only available when "hibernate.search.coordination.strategy" is "outbox-polling".

The default for this value is `HibernateOrmMapperOutboxPollingSettings.Defaults.COORDINATION_ENTITY_MAPPING_AGENT_UUID_GEN_STRATEGY`.

`hibernate.search.coordination.entity.mapping.agent.uuid_type`

The name of the `Hibernate ORM constant type code used to identify a generic SQL type` used for representing an UUID in the agent table.

Only available when "hibernate.search.coordination.strategy" is "outbox-polling".

The default for this value is `HibernateOrmMapperOutboxPollingSettings.Defaults.COORDINATION_ENTITY_MAPPING_AGENT_UUID_TYPE`.

Default value: `"default"`

`hibernate.search.coordination.entity.mapping.outboxevent.catalog`

The database catalog to use for the outbox event table.

Only available when "hibernate.search.coordination.strategy" is "outbox-polling".

Defaults to the default catalog configured in Hibernate ORM. See `MappingSettings.DEFAULT_CATALOG`.

`hibernate.search.coordination.entity.mapping.outboxevent.schema`

The database schema to use for the outbox event table.

Only available when "hibernate.search.coordination.strategy" is "outbox-polling".

Defaults to the default schema configured in Hibernate ORM. See `MappingSettings.DEFAULT_SCHEMA`.

`hibernate.search.coordination.entity.mapping.outboxevent.table`

The name of the outbox event table.

Only available when "hibernate.search.coordination.strategy" is "outbox-polling".

The default for this value is "HSEARCH_OUTBOX_EVENT".

`hibernate.search.coordination.entity.mapping.outboxevent.uuid_gen_strategy`

The name of UUID generator strategy to be used by `UuidGenerator` for the outbox event table.

Only available when "hibernate.search.coordination.strategy" is "outbox-polling".

The default for this value is `HibernateOrmMapperOutboxPollingSettings.Defaults.COORDINATION_ENTITY_MAPPING_OUTBOX_EVENT_UUID_GEN_STRATEGY`.

`hibernate.search.coordination.entity.mapping.outboxevent.uuid_type`

The name of the `Hibernate ORM constant type code used to identify a generic SQL type` used for representing an UUID in the outbox event table.

Only available when "hibernate.search.coordination.strategy" is "outbox-polling".

The default for this value is `HibernateOrmMapperOutboxPollingSettings.Defaults.COORDINATION_ENTITY_MAPPING_OUTBOX_EVENT_UUID_TYPE`.

`hibernate.search.coordination.event_processor.batch_size`

In the event processor, how many outbox events, at most, are processed in a single transaction.

Only available when "hibernate.search.coordination.strategy" is "outbox-polling".

Expects a positive Integer value, such as `50`, or a String that can be parsed into such Integer value.

Defaults to `HibernateOrmMapperOutboxPollingSettings.Defaults.COORDINATION_EVENT_PROCESSOR_BATCH_SIZE`.

Default value: `50`

`hibernate.search.coordination.event_processor.enabled`

Whether the application will process entity change events.

Only available when "hibernate.search.coordination.strategy" is "outbox-polling".

Expects a Boolean value such as `true` or `false`, or a string that can be parsed into a Boolean value.

Defaults to `HibernateOrmMapperOutboxPollingSettings.Defaults.COORDINATION_EVENT_PROCESSOR_ENABLED`.

When the event processor is disabled, events will still be produced by this application node whenever an entity changes, but indexing will not happen on this application node and is assumed to happen on another node.

Default value: `true`

`hibernate.search.coordination.event_processor.event_lock_retry_delay`

How long the event processor must wait before retrying to lock the outbox event database records.

Only available when "hibernate.search.coordination.strategy" is "outbox-polling".

Only applicable when the database supports skipping locked rows when locking.

Expects a positive integer value in seconds, such as `5`, or a String that can be parsed into such Integer value.

Use the value `0` to retry locking events as soon as possible, with no delay.

Defaults to `HibernateOrmMapperOutboxPollingSettings.Defaults.COORDINATION_EVENT_PROCESSOR_EVENT_LOCK_RETRY_DELAY`.

Default value: `2`

`hibernate.search.coordination.event_processor.event_lock_retry_max`

How many times the event processor must try locking the outbox event database records for processing before leaving them to be processed in another batch.

Only available when "hibernate.search.coordination.strategy" is "outbox-polling".

Only applicable when the database supports skipping locked rows when locking.

Expects a positive integer value, such as `10`, or a String that can be parsed into such Integer value.

Defaults to `HibernateOrmMapperOutboxPollingSettings.Defaults.COORDINATION_EVENT_PROCESSOR_EVENT_LOCK_RETRY_MAX`.

Default value: `20`

`hibernate.search.coordination.event_processor.order`

In the event processor, the order in which outbox events are processed.

Only available when "hibernate.search.coordination.strategy" is "outbox-polling".

Expects one of the "external representation" strings defined in `OutboxEventProcessingOrder`.

Defaults to `HibernateOrmMapperOutboxPollingSettings.Defaults.COORDINATION_EVENT_PROCESSOR_ORDER`.

`hibernate.search.coordination.event_processor.polling_interval`

In the event processor, how long to wait for another query to the outbox events table after a query didn't return any event, in milliseconds.

Only available when "hibernate.search.coordination.strategy" is "outbox-polling".

Hibernate Search will wait that long before polling again if the last polling didn't return any event:

- High values mean higher latency between an entity change and the corresponding update in the index, but less stress on the database when there are no events to process.
- Low values mean lower latency between an entity change and the corresponding update in the index, but more stress on the database when there are no events to process.

Expects a positive Integer value in milliseconds, such as `1000`, or a String that can be parsed into such Integer value.

Defaults to `HibernateOrmMapperOutboxPollingSettings.Defaults.COORDINATION_EVENT_PROCESSOR_POLLING_INTERVAL`.

Default value: `100`

`hibernate.search.coordination.event_processor.pulse_expiration`

How long, in milliseconds, an event processor "pulse" remains valid before considering the agent disconnected and forcibly removing it from the cluster.

Only available when "hibernate.search.coordination.strategy" is "outbox-polling".

Every agent registers itself in a database table. Regularly, while polling for events to process, mass indexer agent performs a `"pulse"`: it pauses what it was doing and (among other things) updates its entry in the table, to let other agents know it's still alive and prevent an expiration. If an agent fails to update its entry for longer than the value of the expiration interval, it will be considered disconnected: other agents will forcibly remove its entry from the table, and will perform rebalancing (reassign shards) as necessary.

The expiration interval must be set to a value at least 3 times larger than the `pulse interval`:

- Low values (closer to the pulse interval) mean less time wasted not processing events when a node abruptly leaves the cluster due to a crash or network failure, but increased risk of wasting time not processing events because an event processor is incorrectly considered disconnected.
- High values (much larger than the pulse interval) mean more time wasted not processing events when a node abruptly leaves the cluster due to a crash or network failure, but reduced risk of wasting time not processing events because an event processor is incorrectly considered disconnected.

Expects a positive Integer value in milliseconds, such as `30000`, or a String that can be parsed into such Integer value.

Defaults to `HibernateOrmMapperOutboxPollingSettings.Defaults.COORDINATION_EVENT_PROCESSOR_PULSE_EXPIRATION`.

Default value: `30000`

`hibernate.search.coordination.event_processor.pulse_interval`

How long, in milliseconds, the event processor can poll for events before it must perform a "pulse".

Only available when "hibernate.search.coordination.strategy" is "outbox-polling".

Every agent registers itself in a database table. Regularly, while polling for events to process, the event processor performs a "pulse": it pauses indexing and:

- It updates its own entry in the table, to let other agents know it's still alive and prevent an expiration
- It removes any other agents that expired from the table
- It suspends itself if it notices a mass indexer is running
- It performs rebalancing (reassignment of shards) if the number of agents participating in background indexing changed since the last pulse

The pulse interval must be set to a value between the `polling interval` and one third (1/3) of the `expiration interval`:

- Low values (closer to the polling interval) mean less time wasted not processing events when a node joins or leaves the cluster, and reduced risk of wasting time not processing events because an event processor is incorrectly considered disconnected, but more stress on the database because of more frequent checks of the list of agents.
- High values (closer to the expiration interval) mean more time wasted not processing events when a node joins or leaves the cluster, and increased risk of wasting time not processing events because an event processor is incorrectly considered disconnected, but less stress on the database because of less frequent checks of the list of agents.

Expects a positive Integer value in milliseconds, such as `2000`, or a String that can be parsed into such Integer value.

Defaults to `HibernateOrmMapperOutboxPollingSettings.Defaults.COORDINATION_EVENT_PROCESSOR_PULSE_INTERVAL`.

Default value: `2000`

`hibernate.search.coordination.event_processor.retry_delay`

How long the event processor must wait before re-processing an event after its processing failed.

Only available when "hibernate.search.coordination.strategy" is "outbox-polling".

Expects a positive integer value in seconds, such as `10`, or a String that can be parsed into such Integer value.

Use the value `0` to reprocess the failed events as soon as possible, with no delay.

Defaults to `HibernateOrmMapperOutboxPollingSettings.Defaults.COORDINATION_EVENT_PROCESSOR_RETRY_DELAY`.

Default value: `30`

`hibernate.search.coordination.event_processor.shards.assigned`

The indices of shards assigned to this application node for event processing.

WARNING: shards must be uniquely assigned to one and only one application nodes. Failing that, some events may not be processed or may be processed twice or in the wrong order, resulting in errors and/or out-of-sync indexes.

Only available when "hibernate.search.coordination.strategy" is "outbox-polling".

When this property is set, "hibernate.search.coordination.event_processor.shards.total_count" must also be set.

Expects a shard index, i.e. an Integer value between `0` (inclusive) and the `total shard count` (exclusive), or a String that can be parsed into such shard index, or a String containing multiple such shard index strings separated by commas, or a `Collection<Integer>` containing such shard indices.

No default: must be provided explicitly if you want static sharding.

`hibernate.search.coordination.event_processor.shards.total_count`

The total number of shards across all application nodes for event processing.

WARNING: This property must have the same value for all application nodes, and must never change unless all application nodes are stopped, then restarted. Failing that, some events may not be processed or may be processed twice or in the wrong order, resulting in errors and/or out-of-sync indexes.

Only available when "hibernate.search.coordination.strategy" is "outbox-polling".

When this property is set, "hibernate.search.coordination.event_processor.shards.assigned" must also be set.

Expects an Integer value of at least `2`, or a String that can be parsed into such Integer value.

No default: must be provided explicitly if you want static sharding.

`hibernate.search.coordination.event_processor.transaction_timeout`

In the event processor, the timeout for transactions processing outbox events.

Only available when "hibernate.search.coordination.strategy" is "outbox-polling".

Only effective when a JTA transaction manager is configured.

Expects a positive Integer value in seconds, such as `10`, or a String that can be parsed into such Integer value.

When using JTA and this property is not set, Hibernate Search will use whatever default transaction timeout is configured in the JTA transaction manager.

`hibernate.search.coordination.mass_indexer.polling_interval`

In the mass indexer, how long to wait for another query to the agent table when actively waiting for event processors to suspend themselves, in milliseconds.

Only available when "hibernate.search.coordination.strategy" is "outbox-polling".

Hibernate Search will wait that long before polling again when it finds other agents haven't suspended yet:

- Low values will reduce the time it takes for the mass indexer agent to detect that event processors finally suspended themselves, but will increase the stress on the database while the mass indexer agent is actively waiting.
- High values will increase the time it takes for the mass indexer agent to detect that event processors finally suspended themselves, but will reduce the stress on the database while the mass indexer agent is actively waiting.

Expects a positive Integer value in milliseconds, such as `1000`, or a String that can be parsed into such Integer value.

Defaults to `HibernateOrmMapperOutboxPollingSettings.Defaults.COORDINATION_MASS_INDEXER_POLLING_INTERVAL`.

Default value: `100`

`hibernate.search.coordination.mass_indexer.pulse_expiration`

How long, in milliseconds, a mass indexer agent "pulse" remains valid before considering the agent disconnected and forcibly removing it from the cluster.

Only available when "hibernate.search.coordination.strategy" is "outbox-polling".

Every agent registers itself in a database table. Regularly, while polling for events to process, each agent performs a `"pulse"`: it pauses what it was doing and (among other things) updates its entry in the table, to let other agents know it's still alive and prevent an expiration. If an agent fails to update its entry for longer than the value of the expiration interval, it will be considered disconnected: other agents will forcibly remove its entry from the table, and will resume their work as if the expired agent didn't exist.

The expiration interval must be set to a value at least 3 times larger than the `pulse interval`:

- Low values (closer to the pulse interval) mean less time wasted with event processors not processing events when a mass indexer agent terminates due to a crash, but increased risk of event processors starting to process events again during mass indexing because a mass indexer agent is incorrectly considered disconnected.
- High values (much larger than the pulse interval) mean more time wasted with event processors not processing events when a mass indexer agent terminates due to a crash, but reduced risk of event processors starting to process events again during mass indexing because a mass indexer agent is incorrectly considered disconnected.

Expects a positive Integer value in milliseconds, such as `30000`, or a String that can be parsed into such Integer value.

Defaults to `HibernateOrmMapperOutboxPollingSettings.Defaults.COORDINATION_MASS_INDEXER_PULSE_EXPIRATION`.

Default value: `30000`

`hibernate.search.coordination.mass_indexer.pulse_interval`

How long, in milliseconds, the mass indexer can wait before it must perform a "pulse".

Only available when "hibernate.search.coordination.strategy" is "outbox-polling".

Every agent registers itself in a database table. Regularly, the mass indexer performs a "pulse":

- It updates its owm entry in the table, to let other agents know it's still alive and prevent an expiration
- It removes any other agents that expired from the table

The pulse interval must be set to a value between the `polling interval` and one third (1/3) of the `expiration interval`:

- Low values (closer to the polling interval) mean reduced risk of event processors starting to process events again during mass indexing because a mass indexer agent is incorrectly considered disconnected, but more stress on the database because of more frequent updates of the mass indexer agent's entry in the agent table.
- High values (closer to the expiration interval) mean increased risk of event processors starting to process events again during mass indexing because a mass indexer agent is incorrectly considered disconnected, but less stress on the database because of less frequent updates of the mass indexer agent's entry in the agent table.

Expects a positive Integer value in milliseconds, such as `2000`, or a String that can be parsed into such Integer value.

Defaults to `HibernateOrmMapperOutboxPollingSettings.Defaults.COORDINATION_MASS_INDEXER_PULSE_INTERVAL`.

Default value: `2000`

`hibernate.search.coordination.tenants`

The root property for coordination properties specific to a tenant, e.g. "hibernate.search.coordination.tenants.tenant1.something = somevalue".

A.10. Hibernate Search Mapper - POJO Standalone

`hibernate.search.indexing.mass.default_clean_operation`

The default index cleaning operation to apply during mass indexing, unless configured explicitly.

Expects a `MassIndexingDefaultCleanOperation` value, or a String representation of such value.

Defaults to `StandalonePojoMapperSettings.Defaults.INDEXING_MASS_DEFAULT_CLEAN_OPERATION`.

`hibernate.search.indexing.plan.synchronization.strategy`

How to synchronize between application threads and indexing triggered by the `SearchSession`'s `indexing plan`.

Expects one of the strings defined in `IndexingPlanSynchronizationStrategy`, or a reference to a bean of type `IndexingPlanSynchronizationStrategy`.

Defaults to `StandalonePojoMapperSettings.Defaults.INDEXING_PLAN_SYNCHRONIZATION_STRATEGY`.

`hibernate.search.mapping.build_missing_discovered_jandex_indexes`

Whether Hibernate Search should automatically build Jandex indexes for types registered for annotation processing (entities in particular), to ensure that all "root mapping" annotations in those JARs (e.g. `ProjectionConstructor`) are taken into account.

Expects a Boolean value such as `true` or `false`, or a string that can be parsed into a Boolean value.

Defaults to `StandalonePojoMapperSettings.Defaults.MAPPING_BUILD_MISSING_DISCOVERED_JANDEX_INDEXES`.

Default value: `true`

`hibernate.search.mapping.configurer`

A configurer for the Hibernate Search mapping.

Expects a single-valued or multi-valued reference to beans of type `StandalonePojoMappingConfigurer`.

Defaults to no value.

`hibernate.search.mapping.discover_annotated_types_from_root_mapping_annotations`

Whether Hibernate Search should automatically discover annotated types present in the Jandex index that are also annotated with `root mapping annotations`.

When enabled, if an annotation meta-annotated with `RootMapping` is found in the Jandex index, and a type annotated with that annotation (e.g. `SearchEntity` or `ProjectionConstructor`) is found in the Jandex index, then that type will automatically be scanned for mapping annotations, even if the type wasn't explicitly added.

Expects a Boolean value such as `true` or `false`, or a string that can be parsed into a Boolean value.

Defaults to `StandalonePojoMapperSettings.Defaults.MAPPING_DISCOVER_ANNOTATED_TYPES_FROM_ROOT_MAPPING_ANNOTATIONS`.

Default value: `true`

`hibernate.search.mapping.multi_tenancy.enabled`

Enables or disables multi-tenancy.

If multi-tenancy is enabled, every `session` will need to be assigned a tenant identifier.

Expects a boolean value.

Defaults to `StandalonePojoMapperSettings.Defaults.MULTI_TENANCY_ENABLED`.

Default value: `false`

`hibernate.search.mapping.multi_tenancy.tenant_identifier_converter`

How to convert tenant identifier to and form the string representation.

When multi-tenancy is enabled, and non-string tenant identifiers are used a custom converter **must** be provided through this property.

Defaults to `StandalonePojoMapperSettings.Defaults.MULTI_TENANCY_TENANT_IDENTIFIER_CONVERTER`. This converter only supports string tenant identifiers and will fail if some other type of identifiers is used.

`hibernate.search.schema_management.strategy`

The schema management strategy, controlling how indexes and their schema are created, updated, validated or dropped on startup and shutdown.

Expects a `SchemaManagementStrategyName` value, or a String representation of such value.

Defaults to `StandalonePojoMapperSettings.Defaults.SCHEMA_MANAGEMENT_STRATEGY`.

Appendix B: List of all available logging categories

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

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

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

The following list provides the Hibernate Search logging categories available in the 8.4.0.CR1 version. The information includes the modules using a particular logging category and the levels at which the logger may produce messages. Log categories are grouped in a way that allows multiple categories to be enabled simultaneously by specifying the most common subcategory, where the `org.hibernate.search` is the root category.

`org.hibernate.search`

Description

The root category for any Hibernate Search logs. It may also include logs that do not fit any other, more specific, logging category.

Used in modules

`hibernate-search-engine`, `hibernate-search-util-common`

Produces messages with log levels

`DEBUG`, `ERROR`, `INFO`, `TRACE`, `WARN`

`org.hibernate.search.analysis.lucene`

Description

Logs related to the analysis specific to the Lucene backend. May include information on misconfigured or misbehaving analyzers/normalisers.

Used in modules

`hibernate-search-backend-lucene`

Produces messages with log levels

`WARN`

`org.hibernate.search.configuration`

Description

Logs related to Hibernate Search configuration in general.

Used in modules

`hibernate-search-engine`

Produces messages with log levels

`DEBUG`, `INFO`, `WARN`

`org.hibernate.search.configuration.elasticsearch`

Description

Logs related to the Elasticsearch-specific backend configuration.

Used in modules

`hibernate-search-backend-elasticsearch`

Produces messages with log levels

`DEBUG`, `WARN`

`org.hibernate.search.configuration.lucene`

Description

Logs related to the Lucene-specific backend configuration.

Used in modules

`hibernate-search-backend-lucene`

Produces messages with log levels

DEBUG, WARN

`org.hibernate.search.configuration.mapper.orm`

Description

Logs related to Hibernate ORM mapper-specific configuration.

Used in modules

`hibernate-search-mapper-orm`

Produces messages with log levels

DEBUG

`org.hibernate.search.configuration.mapper.orm.outboxpolling`

Description

Logs related to the outbox polling-specific configuration.

Used in modules

`hibernate-search-mapper-orm-outbox-polling`

Produces messages with log levels

DEBUG

`org.hibernate.search.deprecation.mapper.orm`

Description

Logs related to the usage of deprecated configuration properties or configuration property values specific to the Hibernate ORM mapper.

Used in modules

`hibernate-search-mapper-orm`

Produces messages with log levels

WARN

`org.hibernate.search.deprecation.mapper.orm.outboxpolling`

Description

Logs related to the usage of deprecated configuration properties or configuration property values specific to the outbox polling.

Used in modules

`hibernate-search-mapper-orm-outbox-polling`

Produces messages with log levels

WARN

`org.hibernate.search.elasticsearch`

Description

The main category for the Elasticsearch backend-specific logs. It may also include logs that do not fit any other, more specific, Elasticsearch category.

Used in modules

`hibernate-search-backend-elasticsearch`

Produces messages with log levels

DEBUG

`org.hibernate.search.elasticsearch.client`

Description

Logs information on low-level Elasticsearch backend operations.

This may include warnings about misconfigured Elasticsearch REST clients or index operations.

Used in modules

`hibernate-search-backend-elasticsearch`

Produces messages with log levels

`INFO`

`org.hibernate.search.elasticsearch.client.request`

Description

Logs executed requests and responses sent to the Elasticsearch cluster. It also includes the execution time of the request.

Used in modules

`hibernate-search-backend-elasticsearch`

Produces messages with log levels

`DEBUG`, `TRACE`

`org.hibernate.search.executor`

Description

Logs related to various Hibernate Search internal executors.

Used in modules

`hibernate-search-engine`

Produces messages with log levels

`DEBUG`, `TRACE`

`org.hibernate.search.indexing.mapper.orm`

Description

Logs related to the indexing process that are Hibernate ORM mapper specific.

Used in modules

`hibernate-search-mapper-orm`

Produces messages with log levels

`TRACE`

`org.hibernate.search.loading.mapper.orm`

Description

Logs related to the loading process that are Hibernate ORM mapper specific.

Used in modules

`hibernate-search-mapper-orm`

Produces messages with log levels

`DEBUG`

`org.hibernate.search.lucene`

Description

The main category for the Lucene backend-specific logs. It may also include logs that do not fit any other, more specific, Lucene category.

Used in modules

`hibernate-search-backend-lucene`

Produces messages with log levels

`DEBUG`, `INFO`, `TRACE`, `WARN`

`org.hibernate.search.lucene.infostream`

Description

Logs the Lucene infostream.

To enable the logger, the category needs to be enabled at TRACE level and configuration property `io.writer.infostream` needs to be enabled on the index.

See `org.hibernate.search.backend.lucene.cfg.LuceneIndexSettings.IO_WRITER_INFOSTREAM` for more details.

Used in modules

`hibernate-search-backend-lucene`

Produces messages with log levels

`TRACE`

`org.hibernate.search.mapper.jakarta.batch`

Description

Logs related to the batch indexing operations.

Used in modules

`hibernate-search-mapper-orm-jakarta-batch-core`

Produces messages with log levels

`DEBUG`, `INFO`, `TRACE`, `WARN`

`org.hibernate.search.mapper.massindexing`

Description

Logs related to various mass indexing operations.

Used in modules

`hibernate-search-mapper-pojo-base`

Produces messages with log levels

`DEBUG`, `ERROR`, `INFO`, `WARN`

`org.hibernate.search.mapper.orm`

Description

The main category for the Hibernate ORM mapper-specific logs. It may also include logs that do not fit any other, more specific, Hibernate ORM mapper category.

Used in modules

`hibernate-search-mapper-orm`

Produces messages with log levels

`DEBUG`, `ERROR`, `INFO`, `TRACE`, `WARN`

`org.hibernate.search.mapper.orm.outboxpolling`

Description

The main category for the outbox polling-specific logs. It may also include logs that do not fit any other, more specific, outbox polling category.

Used in modules

`hibernate-search-mapper-orm-outbox-polling`

Produces messages with log levels

`DEBUG`, `INFO`, `TRACE`, `WARN`

`org.hibernate.search.mapper.projection`

Description

Logs related to projection operations.

Used in modules

`hibernate-search-mapper-pojo-base`

Produces messages with log levels

`DEBUG`

`org.hibernate.search.mapping.mapper`

Description

Logs related to creating Hibernate Search mapping.

Used in modules

`hibernate-search-mapper-pojo-base`

Produces messages with log levels

`DEBUG`, `WARN`

`org.hibernate.search.query.elasticsearch`

Description

Logs the Elasticsearch queries that are about to be executed and other query related messages.

Used in modules

`hibernate-search-backend-elasticsearch`

Produces messages with log levels

`TRACE`, `WARN`

`org.hibernate.search.query.lucene`

Description

Logs the Lucene queries that are about to be executed and other query related messages.

Used in modules

`hibernate-search-backend-lucene`

Produces messages with log levels

`TRACE`, `WARN`

`org.hibernate.search.version.elasticsearch`

Description

Logs related to the Elasticsearch/OpenSearch version.

Used in modules

`hibernate-search-backend-elasticsearch`

Produces messages with log levels

`WARN`

`org.hibernate.search.version.orm`

Description

Logs the version of Hibernate Search, when the Hibernate ORM mapper is used.

Used in modules

`hibernate-search-mapper-orm`

Produces messages with log levels

`INFO`