Internals of Hibernate Search

2021-10-18

Table of Contents

This section is intended for new Hibernate Search contributors looking for an introduction to how Hibernate Search works.

Knowledge of the Hibernate Search APIs and how to use them is a requirement to understand this section.

General overview

This section focuses on describing what the different parts of Hibernate Search are at a high level and how they interact with each other.

Hibernate Search internals are split into three parts:

Backends

The backends are where "things get done". They implement common indexing and searching interfaces for use by the mappers through "index managers", each providing access to one index. Examples include the Lucene backend, delegating to the Lucene library, and the Elasticsearch backend, delegating to a remote Elasticsearch cluster.

The word "backend" may refer either to a whole Maven module (e.g. "the Elasticsearch backend") or to a single, central class in this module (e.g. the `ElasticsearchBackend` class implementing the `Backend` interface), depending on context.

Mappers

Mappers are what users see. They "map" the user model to an index, and provide APIs consistent with the user model to perform indexing and searching. For instance the POJO mapper provides APIs that allow to index getters and fields of Java objects according to a configuration provided at boot time.

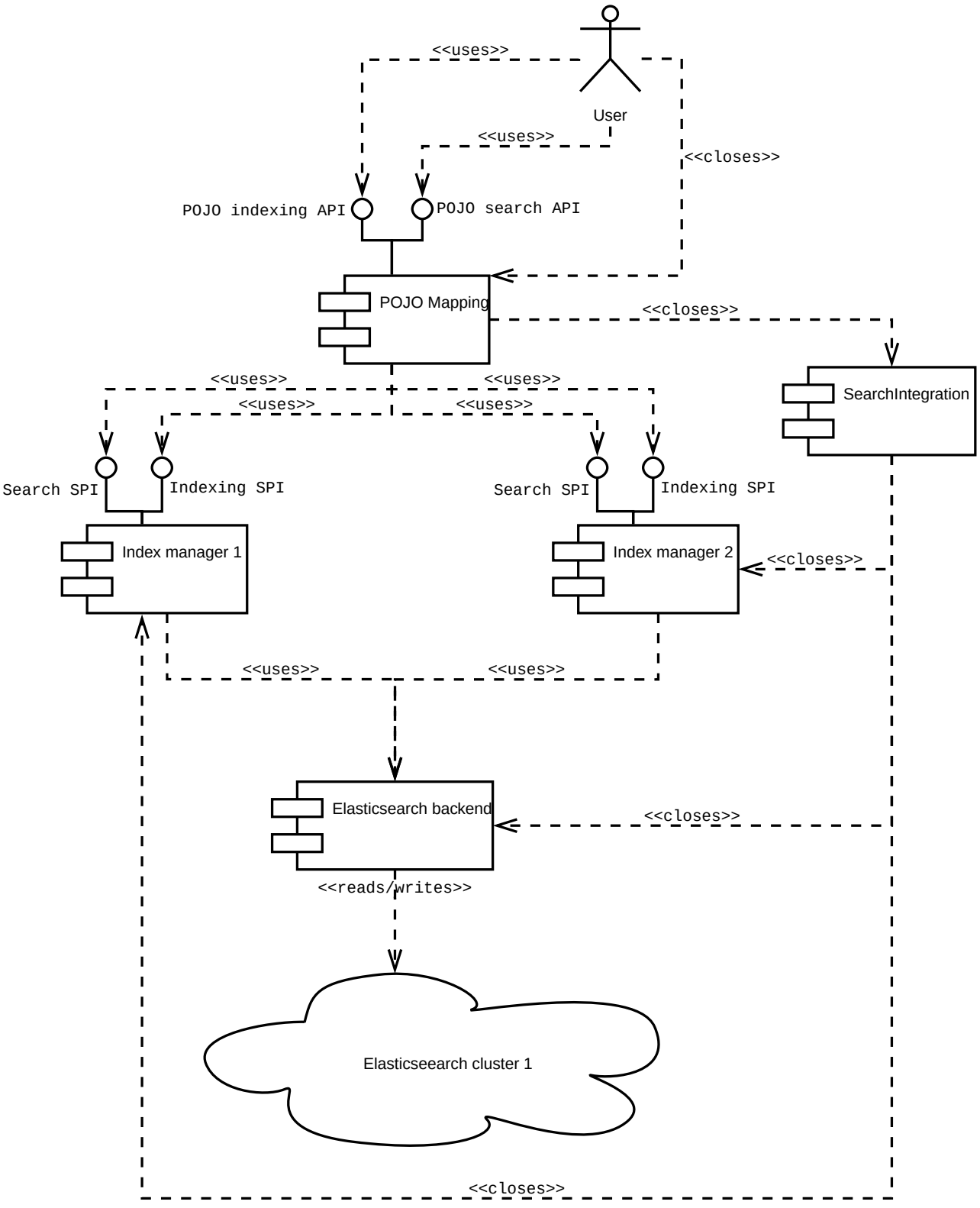
The word "mapper" may refer either to a whole Maven module (e.g. "the POJO mapper") or to a single, central class in this module (e.g. the `PojoMapper` class implementing the `Mapper` interface), depending on context.

Engine

The engine defines some APIs, a lot of SPIs, and implements the code needed to start and stop Hibernate Search, and to "glue" mappers and backends together during bootstrap.

Those parts are strictly separated in order to allow to use them interchangeably. For instance the Elasticsearch backend could be used indifferently with a POJO mapper or a JSON mapper, and we will only have to implement the backend once.

Here is an example of what Hibernate Search would look like at runtime, from a high level perspective:



A "mapping" is a very coarse-grained term, here. A single POJO mapping, for instance, may support many indexed entities.

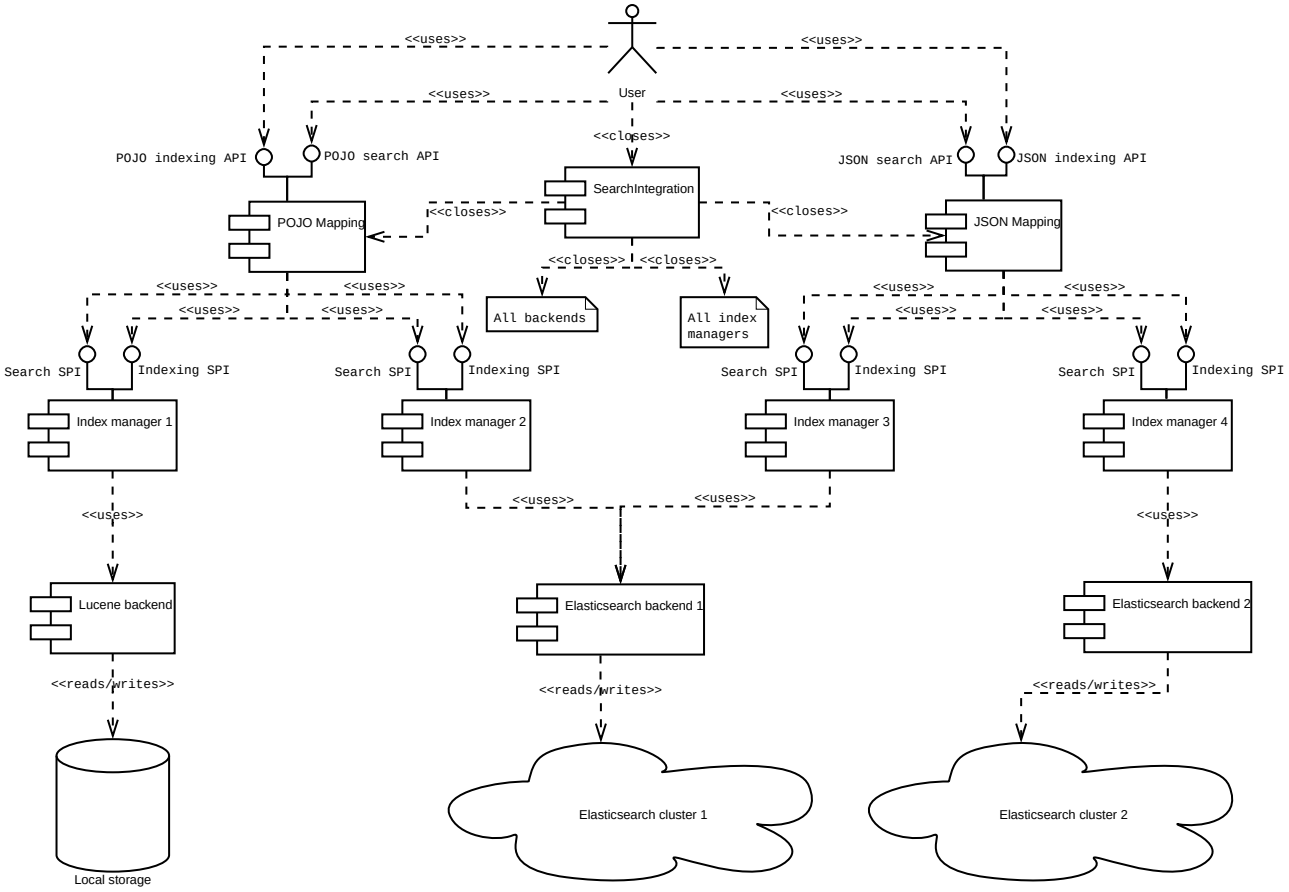
The mapping was provided, during bootstrap, with several "index managers", each exposing SPIs allowing to search and index. The purpose of the mapping is to transform calls to their APIs into call to the index manager SPIs. This requires to perform conversions of:

- indexed data: the data manipulated by the mapping may take any form, but it has to be converted to a document accepted by the index manager.
- index references, e.g. a search query targeting classes `MyEntity` and `MyOtherEntity` must instead target `index manager 1` and `index manager 2`.
- document references, e.g. a search query executed at the index manager level may return "document 1 in index 1 matched the query", but the user wants to see "entity 1 of type MyEntity matched the query".

The purpose of the `SearchIntegration` is mainly to keep track of every resource (mapping or backend) created at bootstrap, and allow to close it all from a single call.

Finally, the purpose of the backend and its index managers is to execute the actual work and return results when relevant.

The architecture is able to support more complex user configurations. The example below shows a Hibernate Search instance with two mappings: a POJO mapping and a JSON mapping.



The example is deliberately a bit contrived, in order to demonstrate some subtleties:

- There are two mappings in this example. Most setups will only configure one mapping, but it is important to keep in mind there may be more. In particular, we anticipate that Infinispan may need multiple different mappings in a single Hibernate Search instance, in order to handle the multiple input types it accepts from its users.

- There are multiple backends in this example. Again, most setups will only ever configure one, but there may be good reasons to use more. For instance if someone wants to index part of the entities in one Elasticsearch cluster, and the other part in another cluster.
- Here, the two mappings each use one index manager from the same Elasticsearch backend. This is currently possible, though whether there are valid uses cases for this remains to be determined, mainly based on the Infinispan needs.

Bootstrap

Bootstrap starts by creating at least two components:

- The `SearchIntegrationBuilder`, which allows to setup all the mapper-independent configuration: bean resolver, configuration property sources for the backends, …
- At least one `MappingInitiator` instance, of a type provided by the mapper module, which will register itself to the `SearchIntegrationBuilder`. From the point of view of the engine, it is a callback that will come into play later.

The idea is that the `SearchIntegrationBuilder` will allow one or more initiators to provide configuration about their mapping, in particular metadata about various "mappable" types (in short, the types manipulated by the user). Then the builder will organize this metadata, check the consistency to some extent, create backends and index manager builders as necessary, and then provide the (organized) metadata back to the mapper module along with handles to index manager builders so that it can start its own bootstrapping.

To sum up: the `SearchIntegrationBuilder` is a facilitator, allowing to start mapper bootstrapping with everything that is necessary:

- engine services and components (`BuildContext`);
- configuration properties (`ConfigurationPropertySource`);
- organized metadata (`TypeMetadataContributorProvider`);
- one handle to the backend layer (`IndexManagerBuildingState`) for each indexed type.

All this is provided to the mapper through the `MappingInitiator` and `Mapper` interfaces.

Mapper bootstrapping is really up to the mapper module, but one thing that won't change is what mappers can do with the handles to the backend layer. These handles are instances of `IndexManagerBuildingState` and each one represents an index manager being built. As the mapper inspects the metadata, it will infer the required fields in the index, and will contribute this information to the backend using the dedicated SPI: `IndexModelBindingContext`, `IndexSchemaElement`, `IndexSchemaFieldContext` are the most important parts.

All this information about the required fields and their options (field type, whether it's stored, how it is

analyzed, ...) will be validated and will allow the backend to build an internal representation of the index schema, which will be used for various, backend-specific purposes, for example initializing a remote Elasticsearch index or inferring the required type of parameters to a range query on a given field.

Indexing

The entry point for indexing is specific to each mapper, and so are the upper levels of each mapper implementation. But at the lower levels, indexing in a mapper comes down to using the backend SPIs.

When indexing, the mapper must build a document that will be passed to the backend. This is done using document elements and index field references. During bootstrap, whenever the mapper declared a field, the backend returned a reference (see `IndexSchemaFieldFinalStep#toReference`). In order to build a document, the mapper extracts data from an object to index, retrieves a document element from the backend, and pass the field reference along with the value to the document element, so that the value is added to the field.

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

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

Index-scale operations such as flush, merge-segments, etc. are unavailable from indexing plans. They are accessed through a separate class, `IndexWorkspace`.

Searching

Searching is a bit different from indexing, in that users are presented with APIs focused on the index rather than the mapped objects. The idea is that when you search, you will mainly target index fields, not properties of mapped objects (though they may happen to have the same name).

As a result, mapper APIs only define entry points for searching so as to offer more natural ways of defining the search scope and to provide additional settings. For example `PojoSearchManager#search` allows to define the search scope using the Java classes of mapped types instead of index names. But somewhere along the API calls, mappers end up exposing generic APIs, for instance `SearchQueryResultDefinitionContext` or `SearchPredicateContainerContext`.

Those generic APIs are mostly implemented in the engine. The implementation itself relies on lower-level, less "user-focused" SPIs implemented by backends, such as `SearchPredicateFactory` or `FieldSortBuilder`.

Also, the SPIs implemented by backends allow mappers to inject a "loading context" (see `SearchQueryBuilderFactory.selectEntity`) that will essentially transform document references into the entity that was initially indexed.

POJO mapper

What we call the POJO mapper is in fact an abstract basis for implementing mappers from Java objects to a full-text index. This module implements most of the necessary logic, and defines SPIs to implement the bits that are specific to each mapper.

There are currently only two implementations: the Hibernate ORM mapper, and the JavaBean mapper. The second one is mostly here to demonstrate that implementing a mapper that doesn't rely on Hibernate ORM is possible: we do not expect much real-life usage.

The following sections do not address everything in the POJO mapper, but instead focus on the more complex parts.

Representation of the POJO metamodel

The bootstrapping process of the POJO mapper relies heavily on the POJO metamodel to infer what will have to be done at runtime. Multiple constructs are used to represent this metamodel.

Models

`PojoTypeModel`, `PojoPropertyModel` and similar are at the root of everything. They are SPIs, to be implemented by the Hibernate ORM mapper for instance, and they provide basic information about mapped types: Java annotations, list of properties, type of each property, "handle" to access each property on an instance of this type, …

Container value extractor paths

`ContainerExtractorPath` and `BoundContainerExtractorPath` both represent a list of `ContainerExtractor` to be applied to a property. They allow to represent what will have to be done to get from a property of type `Map<String, List<MyEntity>>` to a sequence of `MyEntity`, for example. The difference between the "bound" version and the other is that the "bound" version was applied to a POJO model, allowing to guarantee that it will work when applied to that model, and allowing to infer the type of extracted values. See `ContainerExtractorBinder` for more information.

Paths

POJO paths come in two flavors: `PojoModelPath` and `BoundPojoModelPath`. Each has a number of subtypes representing "nodes" in a path. The POJO paths represent how to get from a given type to a given value, by accessing properties, extracting container values (see container value extractor paths above), and casting types. As for container value extractor paths, the difference between the "bound" version and the other is that the "bound" version was applied to a POJO model, allowing to guarantee that it will work when applied to that model (except for casts, obviously), and allowing to infer the type of extracted values.

Additional metadata

`PojoTypeAdditionalMetadata`, `PojoPropertyAdditionalMetadata` and `PojoValueAdditionalMetadata` allow to represent POJO metadata that would not typically be found in a "plain old Java object" without annotations. The metadata may come from various sources: Hibernate Search's annotations, Hibernate Search's programmatic API, or even from other metamodels such as Hibernate ORM's. The "additional metadata" objects are a way to represent this metadata the same way, wherever it comes from. Examples of "additional metadata" include whether a given type is an entity type, property markers ("this property represents a latitude"), or information about inter-entity associations.

Model elements

`PojoModelElement`, `PojoModelProperty` and similar are representations of the POJO metamodel for use by Hibernate Search users in bridges. They are API, on contrary to `PojoTypeModel` et. al. which are SPI, but their implementation relies on both the POJO model and additional metadata. Their main purpose is to shield users from eventual changes in our SPIs, and to allow users to get "accessors" so that they can extract information from the bridge elements at runtime.

When retrieving accessors, users indirectly declare what parts of the POJO model they will extract and use in their bridge, and Hibernate Search actually makes use of this information (see Implicit reindexing resolvers).

Indexing processors

Indexing processors are the objects responsible for extracting data from a POJO and pushing it to a document.

Index processors are organized as trees, each node being an implementation of `PojoIndexingProcessor`. The POJO mapper assigns one tree to each indexed entity type.

Here are the main types of nodes:

- `PojoIndexingProcessorOriginalTypeNode`: A node representing a POJO type (a Java class).
- `PojoIndexingProcessorPropertyNode`: A node representing a POJO property.
- `PojoIndexingProcessorContainerElementNode`: A node representing elements in a container (`List`, `Optional`, ...).

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

- extract data from the Java object passed as input: extract the value of a property, the elements of a list, ...

- pass the extracted data along with the handle to the document being built to a user-configured bridge, which will add fields to the document.
- pass the extracted data along with the handle to the document being built to a nested node, which will in turn "process" its input.

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

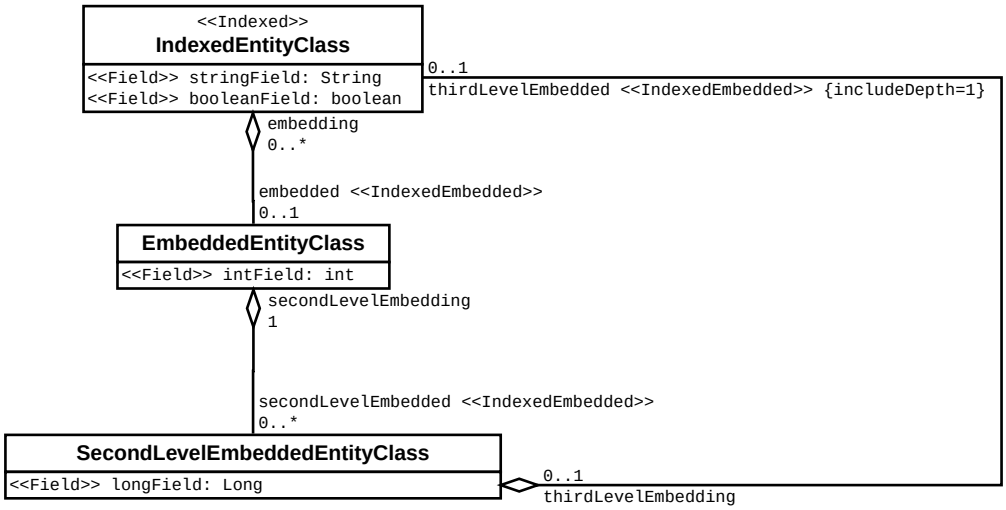
This representation is flexible enough to allow it to represent almost any mapping, simply by defining the appropriate node types and ensuring the indexing processor tree is built correctly, yet explicit enough to not require any metadata lookup at runtime.

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

Bootstrap

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

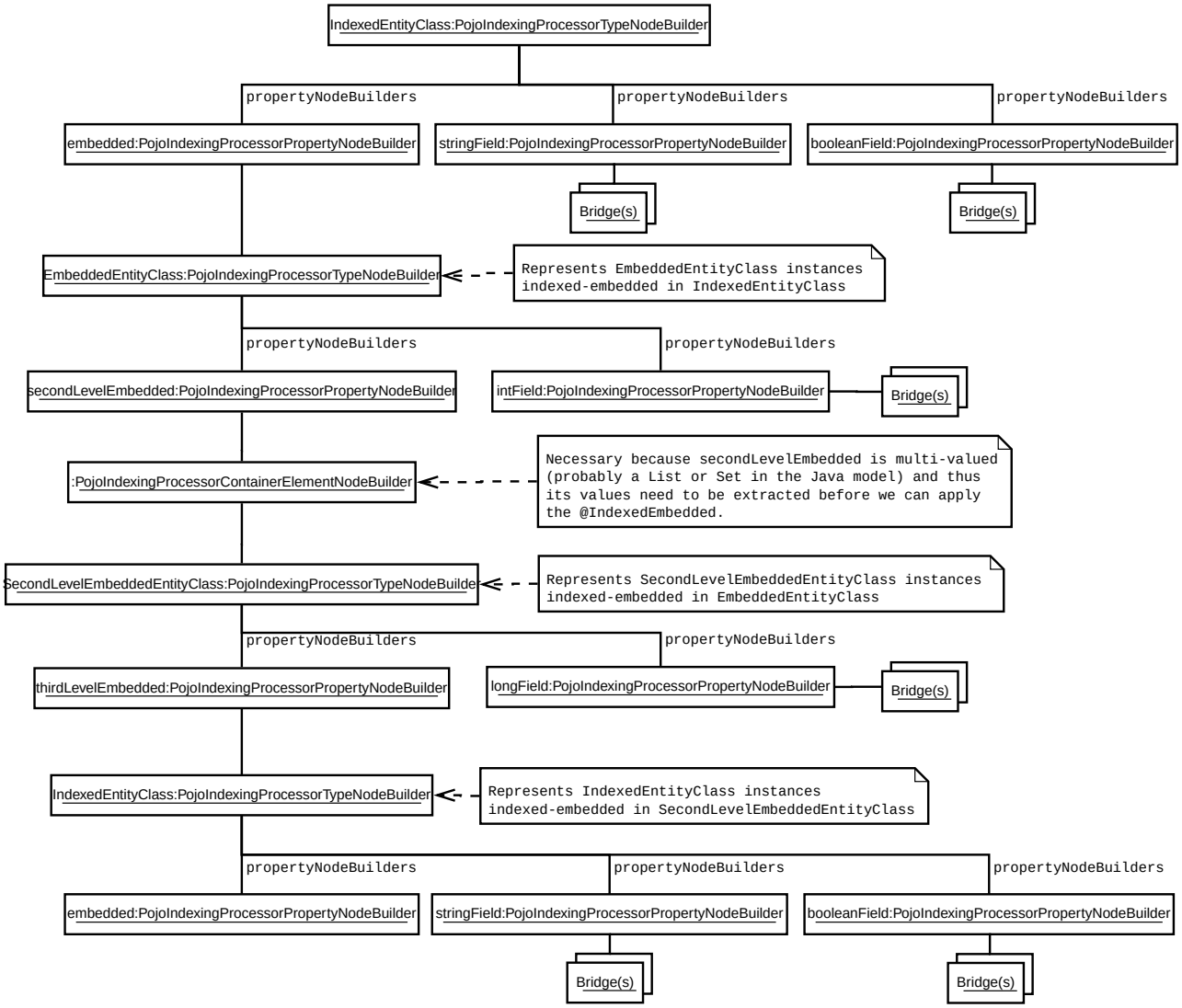
As an example, let's consider the following mapped model:



The class `IndexedEntityClass` is indexed. It has two mapped fields, plus an indexed-embedded on a

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

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



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

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

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

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

Ultimately, the created indexing process tree will follow approximately the same structure as the builder tree. The indexing processor tree may be a bit different from the builder tree, due to optimizations. In particular, some nodes may be trimmed down if we detect that the node will not contribute anything to documents at runtime, which may happen for some property nodes when using `@IndexedEmbedded` with path filtering (`includePaths`) or depth filtering (`includeDepth`).

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

Implicit reindexing resolvers

Reindexing resolvers are the objects responsible for determining, whenever an entity changes, which other entities include that changed entity in their indexed form and should thus be reindexed.

Similarly to indexing processors, the `PojoImplicitReindexingResolver` contains nodes organized as a tree, each node being an implementation of `PojoImplicitReindexingResolverNode`. The POJO mapper assigns one `PojoImplicitReindexingResolver` containing one tree to each indexed or contained entity type. Indexed entity types are those mapped to an index (using `@Indexed` or similar), while "contained" entity types are those being the target of an `@IndexedEmbedded` or being manipulated in a bridge using the `PojoModelElement` API.

Here are the main types of nodes:

- `PojoImplicitReindexingResolverOriginalTypeNode`: A node representing a POJO type (a Java class).
- `PojoImplicitReindexingResolverCastedTypeNode`: A node representing a POJO type (a Java class) to be casted to a supertype or subtype, applying nested nodes only if the cast succeeds.
- `PojoImplicitReindexingResolverPropertyNode`: A node representing a POJO property.
- `PojoImplicitReindexingResolverContainerElementNode`: A node representing elements in a container (`List`, `Optional`, …).
- `PojoImplicitReindexingResolverDirtinessFilterNode`: A node representing a filter, delegating to its nested nodes only if some precise paths are considered dirty.
- `PojoImplicitReindexingResolverMarkingNode`: A node representing a value to be marked

as "to reindex".

At runtime, the root node will be passed the changed entity, the "dirtiness state" of that entity (in short, a list of properties that changed in that entity), and a collector of entities to re-index. Then each node will "resolve" entities to reindex according to its input, i.e. perform one (or more) of the following:

- check that the "dirtiness state" contains specific dirty paths that make reindexing relevant for this node
- extract data from the Java object passed as input: extract the value of a property, the elements of a list, try to cast the object to a given type, …
- pass the extracted data to the collector
- pass the extracted data along with the collector to a nested node, which will in turn "resolve" entities to reindex according to its input.

As with indexing processor, this representation is very flexible, yet explicit enough to not require any metadata lookup at runtime.

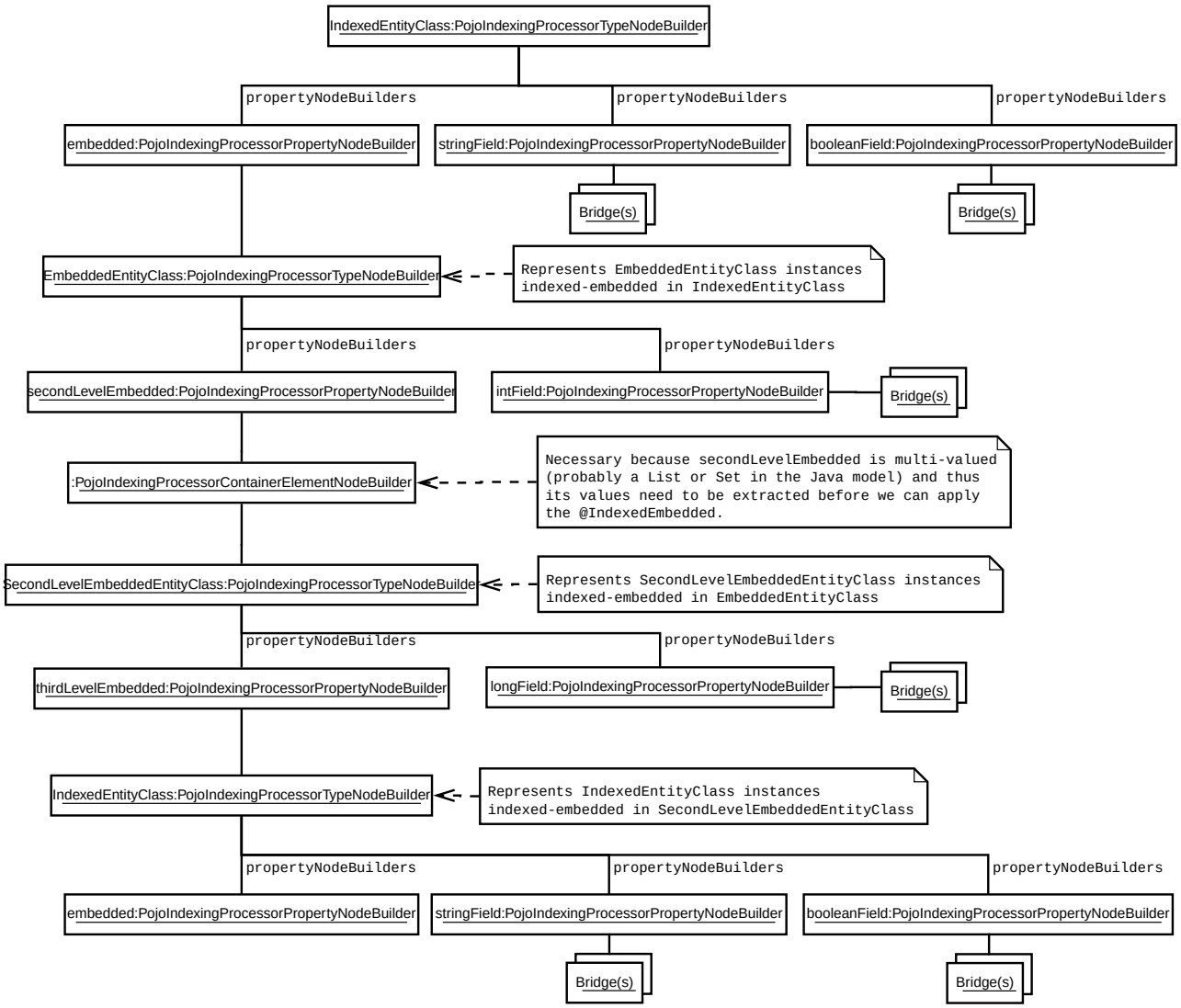
Reindexing resolvers are logged at the debug level during bootstrap. Enable this level of logging for the Hibernate Search classes if you want to understand the reindexing resolver tree that was generated for a given mapping.

Bootstrap

One reindexing resolver tree is built during bootstrap for each indexed or contained type. The entry point to building these resolvers may not be obvious: it is the indexing resolver building process. Indeed, as we build the indexing processor for a given indexed type, we discover all the paths that will be walked through in the entity graph when indexing this type, and thus what the indexed type's indexing process definitely depends on. Which is all the information we need to build the reindexing resolvers.

In order to understand how reindexing resolvers are built, it is important to keep in mind that reindexing resolvers mirror indexing processors: if the indexing processor for entity `A` references entity `B` at some point, then you can be sure that the reindexing resolver for entity `B` will reference entity `A` at some point.

As an example, let's consider the indexing processor builder tree from the previous section (Indexing processors):



As we build the indexing processors, we will also build another tree to represent dependencies from the root type (`IndexedEntityClass`) to each dependency. This is where dependency collectors come into play.

Dependency collectors are organized approximately the same way as the indexing processor builders, as a tree. A root node is provided to the root builder, then one node will be created for each of his children, and so on. Along the way, each builder will be able to notify its dependency collector that it will actually build an indexing processor (it wasn't trimmed down due to some optimization), which means the node needs to be taken into account in the dependency tree. This is done through the `PojoIndexingDependencyCollectorValueNode#collectDependency` method, which triggers some additional steps.

`TypeBridge` and `PropertyBridge` implementations are allowed to go through associations and access properties from different entities. For this reason, when such bridges appear in an indexing processor, we create dependency collector nodes as necessary to model the bridge's dependencies. For more information, see `PojoModelTypeRootElement#contributeDependencies` (type bridges) and `PojoModelPropertyRootElement#contributeDependencies` (property bridges).

Let's see what our dependency collector tree will ultimately look like:

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

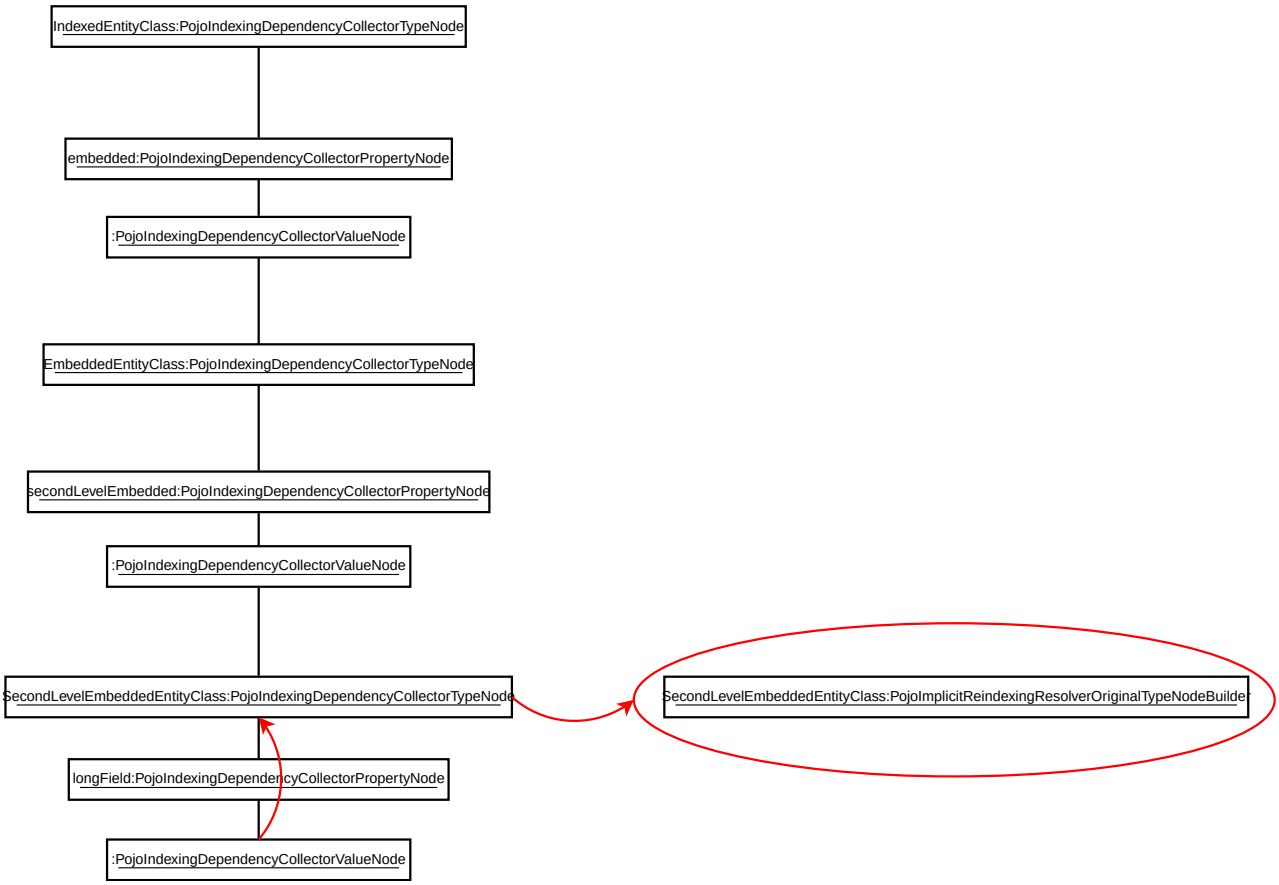
The actual reindexing resolver building happens when `PojoIndexingDependencyCollectorValueNode#collectDependency` is called for each value node. To understand how it works, let us use the value node for `longField` as an example.

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

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

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

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

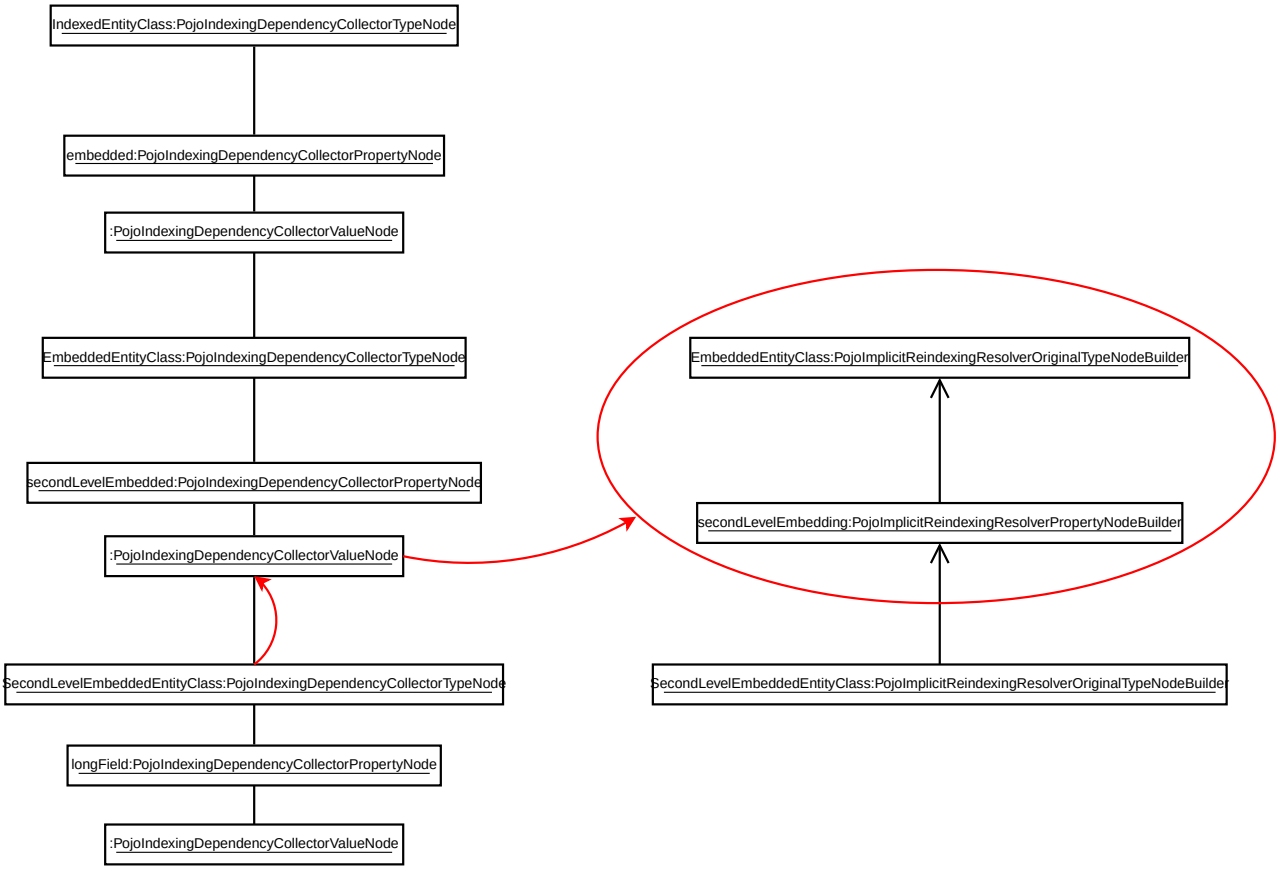


From there, the reindexing resolver builder is passed to the next dependency collector value node using the `PojoIndexingDependencyCollectorValueNode#markForReindexing` method. This method also takes as a parameter the path to the property that is depended on, in this case `longField`.

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

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

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



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

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

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

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

Polymorphism

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

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

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

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

Embedded types

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

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

Fine-grained dirty checking

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

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

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

JSON mapper

The JSON mapper does not currently exist, but there are plans to work on it.