When it comes to associations between entities, things become a bit more complicated compared to the embedded objects we just described. At first sight, associations to embedded objects and associations to entities seem quite similar and, indeed, they are in many ways. The big difference lies in life cycle: embedded objects life cycle is entirely dependent on their owning entity and cannot be referenced by other entities. This is all good and fine for Hibernate Search because when the embedded object is updated, Hibernate Core will raise an event claiming that the owning entity is updated: Hibernate Search just have to update the Lucene document for this entity.

This is not as easy in associations between entities. Remember that using @IndexedEmbedded is essentially a way to denormalize your data and embed the information of two or more entities in a single Lucene document. When an embedded entity is updated, Hibernate Search needs to know if this entity is embedded in an other indexed entity to update its Lucene document: otherwise the denormalized data will be desynchronized (see Figure 1).

![Diagram of Object and Lucene index worlds](image)

**Figure 1** When a change is done on an associated entity, Hibernate Search must update all the documents where the entity is embedded

For Source Code, Free Chapters, the Author Forum and more information about this title go to [http://www.manning.com/bernard](http://www.manning.com/bernard)
Let's take Figure 1 as our working example. When actor is updated, Hibernate Search needs to update the items related to actor. One strategy could be to update all the items in our inventory to make sure everything is up to date. Of course this solution does not fly very far as it would mean to load all the items from the database and reindex them. Instead, Hibernate Search requires you to make the association bidirectional (if not already) and mark the association pointing back to the parent entity with @ContainedBy. Listing 1 shows an example of @ContainedIn usage.

Listing 1 Relations between entities should be bidirectional and the embedded side should be marked with @ContainedBy

```java
@Entity @Indexed
public class Item {
    @ManyToMany
    @IndexedEmbedded
    private Set<Actor> actors;

    @ManyToOne
    @IndexedEmbedded
    private Director director;
    ...
}

@Entity @Indexed
public class Actor {
    @Field private String name;

    @ManyToMany(mappedBy="actors")
    @ContainedIn
    private Set<Item> items;
    ...
}

@Entity @Indexed
public class Director {
    @Id @GeneratedValue @DocumentId private Integer id;
    @Field private String name;

    @OneToMany(mappedBy="director")
    @ContainedIn
    private Set<Item> items;
    ...
}

(1) @ContainedIn is paired to an @IndexedEmbedded annotation on the other side of the relationship.

@ContainedIn can be placed on both collection or single value associations regardless whether or not the association is the owning side of the bidirectional relationship. When Hibernate Search finds a change in an entity having a @ContainedIn association, it marks the associated entity instance(s) as changed: one or more Lucene
document updates will then be triggered. Note that a current limitation in Hibernate Search forces you to mark contained entities as @Indexed. This limitation will be lifted up soon. Some people are confused by @IndexedEmbedded and @ContainedIn: they do not know which side needs to be marked as @IndexedEmbedded to enable the query they want. Think about it that way: @IndexedEmbedded is the property you can navigate to in your queries, @ContainedBy is not.

**NOTE**

Sometimes, it is quite inconvenient to have to make a relationship bidirectional to please Hibernate Search. First of all, if the associated entity does not change in your system (immutable entity), you don't have to add @ContainedIn: since no changes will happen behind Hibernate Search's back your index will be kept synchronized.

If the associated entity changes but you cannot afford a bidirectional relationship, it is always possible to trigger a manual reindexing of the owning entity. If we take Listing 1, you could decide to manually reindex all the item entities every night. Or you could keep track of the actor changes and cherry pick the item entities that need to be reindexed.

It is not uncommon to have embedded entities containing themselves embedded relationships. We can even imagine these embedded relationship pointing to entities having themselves embedded relationships and so on. Embedding too much information in a single Lucene document should be avoided as the indexing time takes longer and the index directory grow bigger. A Lucene document should only contain the necessary bits of information to express planned queries. This poses the question of how to stop embedding associations, at which level to stop.

By default, Hibernate Search stops embedding associations in a given object graph branch when a class has already been processed. Figure 2 describes the default strategy. Hibernate Search raises an exception in this situation to prevent infinite loops create by circular relationships.

![Figure 2](image)

**Exception due to class circularity**

*Figure 2 By default Hibernate Search raises an exception while embedding association when a class has already been processed in a given branch*

The default behavior will not always match your needs. It is common to have:

- A class associated to itself that needs to be indexed (parent child relationships are a good example)
- Entities as both embedded and root indexed entities which leads to indexed object graphs too big. For example, the Actor entity is embedded in the Item entity but is also indexed standalone as we would like to be able to search actors specifically.

@IndexedEmbedded allows you to control the depth at which association embedding stops. By default the depth is not limited and the method described in Figure 1 applies. A depth limit is defined per association: it is the
maximum number of embedding allowed in the branch started by the association (including the current association). Figure 3 is a revised version using explicit depth limit.

Let's explain what is going on in Figure 3. Each association is either marked with an explicit depth or left at the default depth (infinite). The upper branch shows that from the entity A, Hibernate Search is only allowed to embed two associations in depth (in that branch). B is then included. The association between B and C indicates that the depth from this association cannot be higher than three. The association from C to B is not embedded because the maximum number of jumps allowed by the association from A to B was two (one jump from A to B and one jump form B to C). The third branch shows how to limit the depth in a branch involving circular references. The last branch using an infinite depth, shows the default resolution explained by Figure 2.

The depth attribute in @IndexedEmbedded (see Listing 2) is the place to define the maximum depth for a given association:
Listing 2 The maximum depth for embedding associations is defined by @IndexedEmbedded.depth

```java
@Entity @Indexed
public class Item {
    @ManyToMany
    @IndexedEmbedded(depth=4)
    private Set<Actor> actors;

    @ManyToOne
    @IndexedEmbedded(depth=1)
    private Director director;
    ...
}
```

One last warning. Embedding too many associations is a bad practice. To convince you, do not forget that Hibernate Search needs to read the data for all the associated entities. This could load a big object graph if too many associations are marked @IndexedEmbedded.