Hibernate Unidirectional OneToMany and ManyToOne Concepts

OneToMany Unidirectional Mapping

Question:

In this document (scroll down to the Unidirectional section):
http://docs.jboss.org/hibernate/stable/annotations/reference/en/html_single/#entity-mapping-association-collections

It says that a unidirectional one-to-many association with a join table is much preferred to just using a foreign key column in the owned entity. My question is, why is it much preferred?

Answer 1:

Consider the situation where the owned entity type can also be owned by another parent entity type. Do you put foreign key references in the owned table to both parent tables? What if you have three parent types? It just doesn't scale to large designs.

A join-table decouples the join, so that the owned table has no knowledge of the parent table(s), allowing the design to scale elegantly.

Answer 2:

If the child entity has only ever one parent type, then there is no need for a join table. I've done this with JPA (with a hibernate impl.).

Advantages: One less table. Perhaps better performance. No "what is this table for?" type questions.

Disadvantage: From the OO perspective there is an additional dependency between child and parent introduced. In practice this is probably not such a big deal, since the relationship is private in the child.

```
e.g.
parent:
@OneToMany(mappedBy = "parent", cascade = CascadeType.ALL)
@MapKey(name = "name")
private Map children;
child:
@ManyToOne(optional = false)
private Parent parent;
```

How to define Unidirectional OneToMany

JPA 1.0 does not support a unidirectional OneToMany relationship without a JoinTable. JPA 2.0 will have support for a unidirectional OneToMany. In JPA 2.0 a `@JoinColumn` can be used on a OneToMany to define the foreign key, some JPA providers may support this already.

The main issue with an unidirectional OneToMany is that the foreign key is owned by the target object's table, so if the target object has no knowledge of this foreign key, inserting and updating the value is difficult. In a unidirectional OneToMany the source object take ownership of the foreign key field, and is responsible for updating its value.

The target object in a unidirectional OneToMany is an independent object, so it should not rely on the foreign key in any way, i.e. the foreign key cannot be part of its primary key, nor generally have a not null constraint on it.

You can model a collection of objects where the target has no foreign key mapped, but uses it as its primary key, or has no primary key using a Embeddable collection mapping, see **Embeddable Collections.**

If your JPA provider does not support unidirectional OneToMany relationships, then you will need to either add a back reference ManyToOne or a JoinTable. In general it is best to use a JoinTable if you truly want to model a unidirectional OneToMany on the database.

A unidirectional one to many using a foreign key column in the owned entity is not that common and not really recommended. We strongly advise you to use a join table for this kind of association (as explained in the next section). This kind of association is described through a `@JoinColumn`

```
@Entity
public class Customer implements Serializable {
     @OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)
     @JoinColumn(name="CUST_ID")
     public Set<Ticket> getTickets() {
     ...
}
```

```
8
9   @Entity
10  public class Ticket implements Serializable {
11        ... //no bidir
12  }
```

Customer describes a unidirectional relationship with Ticket using the join column CUST_ID.

Unidirectional with join table

A unidirectional one to many with join table is much preferred. This association is described through an `@JoinTable`.

```
1   @Entity
2   public class Trainer {
3         @OneToMany
4         @JoinTable(
5                 name="TrainedMonkeys",
6                 joinColumns = @JoinColumn( name="trainer_id"),
7                 inverseJoinColumns = @JoinColumn( name="monkey_id")
8         )
9         public Set<Monkey> getTrainedMonkeys() {
10        ...
11  }
12
13  @Entity
14  public class Monkey {
15        ... //no bidir
16  }
```

Trainer describes a unidirectional relationship with Monkey using the join table TrainedMonkeys, with a foreign key trainer_id to Trainer (joinColumns) and a foreign key monkey_id to Monkey (inversejoinColumns).

ManyToOne Unidirectional Mapping

Question: What is the difference between Unidirectional and Bidirectional associations? Since the table generated in the db are all the same,so the only difference I found is that each side of the bidiretional assocations will have a refer to the other,and the unidirectional not.

This is a Unidirectional association

```
1   public class User {
2         private int id;
3         private String name;
4         @ManyToOne
5         @JoinColumn(
6                 name = "groupId")
7         private Group group;
8   }
9
10  public class Group {
11        private int id;
12        private String name;
13  }
```

The Bidirectional association

```
1   public class User {
2         private int id;
3         private String name;
4         @ManyToOne
5         @JoinColumn(
6                 name = "groupId")
7         private Group group;
8   }
9   public class Group {
10        private int id;
11        private String name;
12        @OneToMany(mappedBy="group")
13        private List<User> users;
```

```
14  }
```

The difference is whether the group holds a reference of the user. So I wonder if this is the only difference? which is recommended?

Answer 1:

The main differenece is that bidirectional relationship provides navigational access in both directions, so that you can access the other side without explicit queries. Also it allows you to apply cascading options to both directions.

Note that navigational access is not always good, especially for "one-to-very-many" and "many-to-very-many" relationships. Imagine a Group that contains thousands of Users:

- How would you access them? With so many Users, you usually need to apply some filtering and/or pagination, so that you need to execute a query anyway (unless you use collection filtering, which looks like a hack for me). Some developers may tend to apply filtering in memory in such cases, which is obviously not good for performance. Note that having such a relationship can encourage this kind of developers to use it without considering performance implications.
- How would you add new Users to the Group? Fortunately, Hibernate looks at the owning side of relationship when persisting it, so you can only set User.group. However, if you want to keep objects in memory consistent, you also need to add User to Group.users. But it would make Hibernate to fetch all elements of Group.users from the database!

So, I can't agree with the recommendation from the Best Practices. You need to design bidirectional relationships carefully, considering use cases (do you need navigational access in both directions?) and possible performance implications.

Answer 2:

In terms of coding, a bidirectional relationship is more complex to implement because the application is responsible for keeping both sides in synch according to JPA specification 5 (on page 42). Unfortunately the example given in the specification does not give more details, so it does not give an idea of the level of complexity.

When not using a second level cache it is usually not a problem to do not have the relationship methods correctly implemented because the instances get discarded at the end of the transaction.

When using second level cache, if anything gets corrupted because of wrongly implemented relationship handling methods, this means that other transactions will also see the corrupted elements (the second level cache is global).

A correctly implemented bi-directional relationship can make queries and the code simpler, but should not be used if it does not really make sense in terms of business logic.