JPA implementation patterns: Bidirectional Assocations

by Vincent Partington MVB · Aug. 13, 09 · Java Zone · Not set

JPA offers the @OneToMany, @ManyToOne, @OneToOne, and @ManyToMany annotations to map associations between objects. While EJB 2.x offered container managed relationships to manage these associations, and especially to keep bidirectional associations in sync, JPA leaves more up to the developer.

The setup

Let's start by expanding the Order example from the previous blog with an OrderLine object. It has an id, a description, a price, and a reference to the order that contains it:

```
@Entity
public class OrderLine {
        @Id
        @GeneratedValue
        private int id;

        private String description;

        private int price;

        @ManyToOne
        private Order order;

        public int getId() { return id; }
        public void setId(int id) { this.id = id; }
        public void setDescription(String description) { this.description = description; }

        public int getPrice() { return price; }
        public void setPrice(int price) { this.price = price; }

        public Order getOrder() { return order; }
        public void setOrder(Order order) { this.order = order; }
}
```

Using the generic DAO pattern, we quickly get ourselves a very basic OrderLineDao interface and an implementation:

```
public interface OrderLineDao extends Dao<Integer, OrderLine> {
        public List<OrderLine> findOrderLinesByOrder(Order o);
}

public class JpaOrderLineDao extends JpaDao<Integer, OrderLine> implements
                OrderLineDao {

        public List<OrderLine> findOrderLinesByOrder(Order o) {
                Query q = entityManager.createQuery("SELECT e FROM "
                                + entityClass.getName() + " e WHERE order = :o");
                q.setParameter("o", o);
                return (List<OrderLine>) q.getResultList();
        }
}
```

We can use this DAO to add a orderline to an order, or to find all the order lines for an order:

```
        OrderLine line = new OrderLine();
        line.setDescription("Java Persistence with Hibernate");
        line.setPrice(5999);
        line.setOrder(o);
        orderLineDao.persist(line);

        Collection<OrderLine> lines = orderLineDao.findOrderLinesByOrder(o);
```

Mo associations, mo problems

All this is pretty straight forward, but it gets interesting when we make this association bidirectional. Let's add an orderLines field to our Order object and include a naïve implementation of the getter/setter pair:

```
@OneToMany(mappedBy = "order")
        private Set<OrderLine> orderLines = new HashSet<OrderLine>();

        public Set<OrderLine> getOrderLines()
       { return orderLines; }

        public void setOrderLines(Set<OrderLine> orderLines)
       { this.orderLines = orderLines; }
```

The mappedBy field on the @OneToMany annotation tells JPA that this is the reverse side of an association and, instead of mapping this field directly to a database column, it can look at order field of a OrderLine object to know with which Order object it goes.

So without changing the underlying database we can now retrieve the orderlines for an order like this:

```
        Collection<OrderLine> lines = o.getOrderLines();
```

No more need to access the OrderLineDao. 😃

But there is a catch! While container managed relationships (CMR) as defined by EJB 2.x made sure that adding an OrderLine object to the orderLines property of an Order *also* sets the order property on that OrderLine (and vice versa), JPA (being a POJO framework) performs no such magic. This is actually a good thing because it makes our domain objects usable outside of a JPA container, which means you can test them more easily and use them when they have not been persisted (yet). But it can also be confusing for people that were used to EJB 2.x CMR behaviour.

If you run the examples above in separate transactions, you will find that they run correctly. But if you run them within one transaction like the code below does, you will find that the item list while be empty:

```
Order o = new Order();
        o.setCustomerName("Mary Jackson");
        o.setDate(new Date());

        OrderLine line = new OrderLine();
        line.setDescription("Java Persistence with Hibernate");
        line.setPrice(5999);
        line.setOrder(o);
        System.out.println("Items ordered by " + o.getCustomerName() + ": ");
        Collection<OrderLine> lines = o.getOrderLines();
        for (OrderLine each : lines) {
                System.out.println(each.getId() + ": " + each.getDescription()
                                + " at $" + each.getPrice());
        }
```

This can be fixed by adding the following line before the first System.out.println statement:

```
        o.getOrderLines().add(line);
```

Fixing and fixing...

It works, but it's not very pretty. It breaks the abstraction and it's brittle as it depends on the user of our domain objects to correctly invoke these setters and adders. We can fix this by moving that invocation into the definition of OrderLine.setOrder(Order):

```
        public void setOrder(Order order) {
                this.order = order;
                order.getOrderLines().add(this);
        }
```

When can do even better by encapsulating the orderLines property of the Order object in a better manner:

```
public Set<OrderLine> getOrderLines() { return orderLines; }
        public void addOrderLine(OrderLine line) { orderLines.add(line); }
```

And then we can redefine OrderLine.setOrder(Order) as follows:

```
public void setOrder(Order order) {
                this.order = order;
                order.addOrderLine(this);
        }
```

Still with me? I hope so, but if you're not, please try it out and see for yourself.

Now another problem pops up. What if someone directly invokes the Order.addOrderLine(OrderLine) method? The OrderLine will be added to the orderLines collection, but its order property will not point to the order it belongs. Modifying Order.addOrderLine(OrderLine) like below will not work because it will cause an infinite loop with addOrderLine invoking setOrder invoking addOrderLine invoking setOrder etc.:

```
public void addOrderLine(OrderLine line) {
		orderLines.add(line);
		line.setOrder(this);
	}
```

This problem can be solved by introducing an Order.internalAddOrderLine(OrderLine) method that only adds the line to the collection, but does not invoke line.setOrder(this). This method will then be invoked from OrderLine.setOrder(Order) and not cause an infinite loop. Users of the Order class should invoke Order.addOrderLine(OrderLine).

The pattern

Taking this idea to its logical conclusion we end up with these methods for the OrderLine class:

```

	public Order getOrder() { return order; }

	public void setOrder(Order order) {
		if (this.order != null) { this.order.internalRemoveOrderLine(this); }
		this.order = order;
		if (order != null) { order.internalAddOrderLine(this); }
	}
```

And these methods for the Order class:

```
	public Set<OrderLine> getOrderLines() { return Collections.unmodifiableSet(orderLines); }

	public void addOrderLine(OrderLine line) { line.setOrder(this); }
	public void removeOrderLine(OrderLine line) { line.setOrder(null); }

	public void internalAddOrderLine(OrderLine line) { orderLines.add(line); }
	public void internalRemoveOrderLine(OrderLine line) { orderLines.remove(line); }
```

These methods provide a POJO-based implementation of the CMR logic that was built into EJB 2.x. With the typical POJOish advantages of being easier to understand, test, and maintain.

Of course there are a number of variations on this theme:

- If Order and OrderLine are in the same package, you can give the internal... methods package scope to prevent them from being invoked by accident. (This is where C++'s friend class concept would come in handy. Then again, let's not go there. 😉).
- You can do away with the removeOrderLine and internalRemoveOrderLine methods if order lines will never be removed from an order.
- You can move the responsibility for managing the bidirectional association from the OrderLine.setOrder(Order) method to the Order class, basically flipping the idea around. But that would mean spreading the logic over the addOrderLine and removeOrderLine methods.
- Instead of, or in addition to, using Collections.singletonSet to make the orderLine set read-only at run-time, you can also use generic types to make it read-only at compile-time:

  ```
  public Set<? extends OrderLine> getOrderLines() { return Collections.unmodifiableSet(orderLines); }
  ```

  But this makes it harder to mock these objects with a mocking framework such as EasyMock.

There are also some things to consider when using this pattern:

- Adding an OrderLine to an Order does not automatically persist it. You'll need to also invoke the persist method on its DAO (or the EntityManager) to do that. Or you can set the cascade property of the @OneToMany annotation on the Order.orderLines property to CascadeType.PERSIST (at least) to achieve that. More on this when we discuss the EntityManager.persist method.
- Bidirectional associations do not play well with the EntityManager.merge method. We will discuss this when we get to the subject of detached objects.
- When an entity that is part of a bidirectional associated is (about to be) removed, it should also be removed from the other end of the association. This will also come up when we talk about the EntityManager.remove method.
- The pattern above only works when using field access (instead of property/method access) to let your JPA provider populate your entities. Field access is used when the @Id annotation of your entity is placed on the corresponding field as opposed of the corresponding getter. Whether to prefer field access or property/method access is a contentious issue to which I will return in a later blog.
- And last but not least; while this pattern may be a technically sound POJO-based implementation of managed associations, you can argue why you need all those getters and setters. Why would you need to be able to use both Order.addOrderLine(OrderLine) and OrderLine.setOrder(Order) to achieve the same result? Doing away with one of these could make our code simpler. See for example James Holub's article on getters and setters. Then again, we've found that this pattern gives developers that use these domain objects the flexibility to associate them as they wish.

Well, that wraps it up for today. I am very interested to hear your feedback on this and to hear how you manage bidirectional associations in your JPA domain objects. And of *course: see you at the next pattern!*

From http://blog.xebia.com