JPA Implementation Patterns: Data Access Objects

Learn about Data Access Objects (DAO's), their patterns and how to write them.

by Vincent Partington MVB · **Jul. 22, 09 · Java Zone · Tutorial**

The JPA, short for Java Persistence API, is part of the Java EE 5 specification and has been implemented by Hibernate, TopLink, EclipseLink, OpenJPA, and a number of other object-relational mapping (ORM) frameworks. Because JPA was originally designed as part of the EJB 3.0 specification, you can use it within an EJB 3.0 application. But it works equally well outside of EJB 3.0, for example in a Spring application. And when even Gavin King, the designer of Hibernate, recommends using JPA in the second edition of Hibernate in Action, a.k.a. Java Persistence with Hibernate, it's obvious that JPA is here to stay.

Once you get over your fear of annotations ;-), you find that there is plenty of literature out there that explains the objects and methods within the API, the way these objects work together and how you can expect them to be implemented. And when you stick to hello-world-style programs, it all seems pretty straight forward. But when you start writing your first ***real*** *application, you find that things are not so simple. The abstraction provided by JPA is pretty leaky and has ramifications for larger parts of your application than just your Data Access Objects (DAO's) and your domain objects. You need to make decisions on how to handle transactions, lazy loading, detached object (think web frameworks), inheritance, and more. And it turns out that the books and the articles don't really help you here.*

A:link {color: blue;} A:visited {color: #A000A0;} html body p { line-height: 120%; } ul li, ul.menu li, .item-list ul li, li.leaf { list-style-type: disc; list-style-position:outside;} ol li, ol.menu li, .item-list ol li, li.leaf { list-style-type: decimal; list-style-position:outside;}

At least, that is what I discovered when I really started using JPA for the first time. In the coming weeks, I would like to discuss the choices I came up against and the decisions I made and why I made them. When I'm done, we'll have a number of what I would like to not-too-modestly call JPA implementation patterns.

Do we really need a DAO?

So, let's start with the thing you would probably write first in your JPA application: the data access object (DAO). An interesting point to tackle before we even start is whether you

even need a DAO when using JPA. The conclusion of that discussion more than a year ago was "It depends" and while it is very hard to argue with such a conclusion :-), I would like to stick with the idea that a DAO *does* have its place in a JPA application. Arguably it provides only a thin layer on top of JPA, but more importantly making a DAO per entity type gives you these advantages:

- Instead of having to pick the right EntityManager method every time you want to store or load data, you decide which one to use once and you and your whole team can easily stick to that choice.
- You can disallow certain operations for certain entity types. For example, you might never want your code to remove log entries. When using DAO's, you just do not add a remove method to your LogEntry DAO.
- Theoretically, by using DAO's you could switch to another persistence system (like plain JDBC or iBATIS). But because JPA is such a leaky abstraction I think that that is not realistically possible for even a slightly complex application. You *do* get a single point of entry where you can add tracing features or keep performance statistics.
- You can centralize all the queries on a certain entity type instead of scattering them through your code. You could use named queries to keep the queries with the entity type, but you'd still need some central place where the right parameters are set. Putting both the query, the code that sets the parameters, and the cast to the correct return type in the DAO seems a simpler thing to do. For example:

```
public List<ChangePlan> findExecutingChangePlans() {      Query query = entityManager.createQuery(            "SELECT plan FROM ChangePlan plan where plan.sta
```

So when you decide you *are* going to use DAO's, how do you go about writing them? The highlighted (in bold) comment in the Javadoc for Spring's JpaTemplate seems to suggest that there's not much point in using that particular class, which also makes JpaDaoSupport superfluous. Instead you can write your JPA DAO as a POJO using the @PersistenceContext annotation to get an EntityManager reference. It will work in an EJB 3.0 container *and* it will work in Spring 2.0 and up if you add the PersistenceAnnotationBeanPostProcessor bean to your Spring context.

The type-safe generic DAO pattern

Because each DAO shares a lot of functionality with the other DAO's, it makes sense to have a base class with the shared functionality and then subclass from that for each specific DAO. There are alotofblogs out there about such a *type-safe generic DAO* pattern and you can even download some code from Google Code. When we combine elements from all these sources, we get the following JPA implementation pattern for DAO's.

The entity class

Let's say we want to persist the following Order class:

```
@Entity@Table(name = "ORDERS")public class Order {@Id@GeneratedValueprivate int id;private String customerName;private Date date; public int getId() { return id; }
```

Don't worry too much about the details of this class. We will revisit the specifics in other JPA implementation patterns. The @Table annotation is there because ORDER is a reserved keyword in SQL.

The DAO interfaces

First we define a generic DAO interface with the methods we'd like all DAO's to share:

```
public interface Dao<K, E> {      void persist(E entity);      void remove(E entity);      E findById(K id);}
```

The first type parameter, K, is the type to use as the key and the second type parameter, E, is the type of the entity. Next to the basic persist, remove, and findById methods, you might also like to add a List findAll() method. But like the entity class itself, we will revisit the DAO methods in later JPA implementation patterns.

Then we define one subinterface for each entity type we want to persist, adding any entity specific methods we want. For example, if we'd like to be able to query all orders that have been added since a certain date, we can add such a method:

```
public interface OrderDao extends Dao<Integer, Order> {List<Order> findOrdersSubmittedSince(Date date);}
```

The base DAO implementation

The third step is to create a base JPA DAO implementation. It will have basic implementation of all the methods in the standard Dao interface we created in step 1:

```
public abstract class JpaDao<K, E> implements Dao<K, E> {protected Class<E> entityClass; @PersistenceContextprotected EntityManager entityManager; public JpaDao()
```

Most of the implementation is pretty straight forward. Some points to note though:

- The constructor of the JpaDao includes the method proposed by my colleague Arjan Blokzijl to use reflection to get the entity class.
- The @PersistenceContext annotation causes the EJB 3.0 container or Spring to inject the entity manager.
- The entityManager and entityClass fields are protected so that subclasses, i.e. specific DAO implementations, can access them.

The specific DAO implementation

And finally we create such a specific DAO implementation. It extends the basic JPA DAO class *and* implements the specific DAO interface:

```
public class JpaOrderDao extends JpaDao<Integer, Order> implements OrderDao {public List<Order> findOrdersSubmittedSince(Date date) {Query q = entityManager.create(
```

Using the DAO

How you get a reference to an instance of your OrderDao depends upon whether we use EJB 3.0 or Spring. In EJB 3.0 we'd use a annotation like this:

```
@EJB(name="orderDao")private OrderDao orderDao;
```

while in Spring we can use the XML bean files or we can use autowiring like this:

```
@Autowiredpublic OrderDao orderDao;
```

In any case, once we have a reference to the DAO we can use it like this:

```
Order o = new Order();o.setCustomerName("Peter Johnson");o.setDate(new Date());orderDao.persist(o);
```

But we can also use the entity specific query we added to the OrderDao interface:

```
List<Order> orders = orderDao.findOrdersSubmittedSince(date);for (Order each : orders) {System.out.println("order id = " + each.getId());}
```

With this type-safe DAO pattern we get the following advantages:

- No direct dependency on the JPA api from client code.
- Type-safety through the use of generics. Any casts that still need to be done are handled in the DAO implementation.
- One logical place to group all entity-specific JPA code.
- One location to add transaction markers, debugging, profiling, etc. Although as we will see later, we will need to add transaction markers in other parts of our applications too.
- One class to test when testing the database access code. We will revisit this subject in a later JPA implementation pattern.

 I hope this convinces you that you *do* need DAO's with JPA.

And that wraps up the first JPA implementation pattern. In the next blog of this series we will build on this example to discuss the next pattern. In the meantime I would love to hear from you how you write your DAO's!

From http://blog.xebia.com

How do you break a Monolith into Microservices at Scale? This ebook shows strategies and techniques for building scalable and resilient microservices.

Like This Article? Read More From DZone

Database Interaction with DAO and DTO Design Patterns

JPA Implementation Patterns: Saving (Detached) Entities

Using spwrap to Simplify Calling Stored Procedures From Java

Free DZone Refcard

Getting Started With Vaadin 10

Topics: JAVA , JPA , DAO

Opinions expressed by DZone contributors are their own.

Build vs Buy a Data Quality Solution: Which is Best for You?
Melissa Data