The Only DAO You'll Ever Need

By Roger Keays, 29 May 2007

The Data Access Object (DAO) is a pattern which, I believe, is way overused. Basically the idea is to delegate all your persistence operations to a wrapper object to make it easier to change persistence technologies. In a webapp the DAO serves the dual purpose of being a container for objects and persistence operations which can be invoked on them. For this reason, people often extend the pattern to have one DAO per entity, so `Customer` has a `CustomerDAO`, `Product` a `ProductDAO` etc etc etc.

This seemed like overkill to me. What I really wanted is to be able to write a webapp which consisted (in it's entirety) only of entity classes and CRUD templates for those entities.

To achieve this, I created an EntityDAO class with generic add, update and delete methods which are invoked by a JSF <h:commandButton/>. These methods simply call the corresponding persistence operation on the contained object:

```
public class EntityDAO {
    EntityManager em;
    Object current;

    public void addEntity(ActionEvent e) {
        em.getTransaction().begin();
        em.persist(current);
        em.getTransaction().commit();
    }

    public void updateEntity(ActionEvent e) {
        em.getTransaction().begin();
        setCurrent(em.merge(current));
        em.getTransaction().commit();
    }
   ...
}
```

This isn't rocket science, and it doesn't allow you to manage tricky persistence relationships, but it is pretty useful a lot of the time. The one requirement that this idea relies on, is that there is some mechanism to set the EntityDAO.current object as required. This can be done a few different ways. For new objects, you can use the managed bean facility:

```
<h:commandButton value="Create Customer">
  <f:setPropertyActionListener target="${entities.current}" value="${newCustome
  <f:actionListener type="seamless.actions.AddEntity"/>
</h:commandButton>
```

In this example, ${newCustomer} is a managed bean, and the AddEntity ActionListener is a wrapper for the addEntity() method on the EntityDAO. It just allows us to chain ActionListeners.

Update and delete operations are a little more difficult though, because you need to make sure a specific object is set as current, not just any old new one. This is one reason why people use separate DAOs for each class. They just shove them in the session scope to keep a reference to the object to be operated on. This fails with tabbed browsing of course.

For these sorts of conversation-scoped operations, I've implemented a <s:restoreEntity/> component which renders a <input type="hidden"/> field in the form. It differs from a standard <h:inputHidden/> because it applies its value during the Apply Request Values phase. This is earlier in the lifecycle than the standard component which only sets its value during the Update Model phase. It is necessary because other components may rely on the EntityDAO.current being available early in the lifecycle.

Your update form could now look something like this:

```
<h:form>
  <s:restoreEntity/>
  Name:  <h:inputText value="${entities.current.name}"/><br/>
  Phone: <h:inputText value="${entities.current.phone}"/>

  <h:commandButton value="Update" actionListener="${entities.updateEntity}"/>
</h:form>
```

The <s:restoreEntity/> component relies on generic converter for JPA entities to marshal the entity to and from a string, but this will be the topic of another blog because its a handy feature itself.

All this code is available through the Furnace Webapp Framework if you want to use it for yourself. There is also an example webapp using the Entity DAO available for download.

About Roger Keays

Roger Keays is an artist, an engineer, and a student of life. He has no fixed addressand has leftfootprints on 40-something different countries around the world. Roger is addicted to surfing. His other interests are music, psychology, languages, the proper use of semicolons, and finding good food.