Generic Data Access Objects

版本 2

由 Anthony Patricio 于 2009-8-11 上午5:21创建，最后由 Steve Ebersole 于 2010-3-20 下午1:24修改。

This is a pattern for Data Access Objects with JDK 5.0, from the CaveatEmptor example application. It is also explained in the book Java Persistence with Hibernate . Two links you might find useful: Sessions and transactions and Open Session in View .
This time I based the DAO example on interfaces. Tools like Hibernate already provide database portability, so persistence layer portability shouldn't be a driving motivation for interfaces. However, DAO interfaces make sense in more complex applications, when several persistence services are encapsulate in one persistence layer. I'd say that you should use Hibernate (or Java Persistence APIs) directly in most cases, the best reason to use an additional DAO layer is higher abstraction (e.g. methods like getMaximumBid() instead of session.createQuery(...) repeated a dozen times).

The DAO interfaces

I use one interface per persistent entity, with a super interface for common CRUD functionality:

```
01. public interface GenericDAO<T, ID extends Serializable> {
02.
03.     T findById(ID id, boolean lock);
04.
05.     List<T> findAll();
06.
07.     List<T> findByExample(T exampleInstance);
08.
09.     T makePersistent(T entity);
10.
11.     void makeTransient(T entity);
12. }
```

You can already see that this is going to be a pattern for a state-oriented data access API, with methods such as makePersistent() and makeTransient(). Furthermore, to implement a DAO you have to provide a type and an identifier argument. As for most ORM solutions, identifier types have to be serializable.
The DAO interface for a particular entity extends the generic interface and provides the type arguments:

```
01. public interface ItemDAO extends GenericDAO<Item, Long> {
02.
03.     public static final String QUERY_MAXBID = "ItemDAO.QUERY_MAXBID";
04.     public static final String QUERY_MINBID = "ItemDAO.QUERY_MINBID";
05.
```

```
    Bid getMaxBid(Long itemId);
    Bid getMinBid(Long itemId);

}
```

We basically separate generic CRUD operations and actual business-related data access operations from each other. (Ignore the named query constants for now, they are convenient if you use annotations.) However, even if only CRUD operations are needed for a particular entity, you should still write an interface for it, even it it is going to be empty. It is important to use a concrete DAO in your controller code, otherwise you will face some refactoring once you have to introduce specific data access operations for this entity.

An implementation with Hibernate

An implementation of the interfaces could be done with any state-management capable persistence service. First, the generic CRUD implementation with Hibernate:

```
public abstract class GenericHibernateDAO<T, ID extends Serializable>
        implements GenericDAO<T, ID> {

    private Class<T> persistentClass;
    private Session session;

    public GenericHibernateDAO() {
        this.persistentClass = (Class<T>) ((ParameterizedType) getClass()
                                .getGenericSuperclass()).getActualTypeArguments()[0];
     }

    @SuppressWarnings("unchecked")
    public void setSession(Session s) {
        this.session = s;
    }

    protected Session getSession() {
        if (session == null)
            throw new IllegalStateException("Session has not been set on DAO before usage");
        return session;
    }

    public Class<T> getPersistentClass() {
        return persistentClass;
    }

    @SuppressWarnings("unchecked")
    public T findById(ID id, boolean lock) {
        T entity;
        if (lock)
            entity = (T) getSession().load(getPersistentClass(), id, LockMode.UPGRADE);
        else
            entity = (T) getSession().load(getPersistentClass(), id);

        return entity;
    }

    @SuppressWarnings("unchecked")
    public List<T> findAll() {
        return findByCriteria();
    }

    @SuppressWarnings("unchecked")
    public List<T> findByExample(T exampleInstance, String[] excludeProperty) {
        Criteria crit = getSession().createCriteria(getPersistentClass());
        Example example =  Example.create(exampleInstance);
        for (String exclude : excludeProperty) {
```

```
            example.excludeProperty(exclude);
        }
        crit.add(example);
        return crit.list();
    }

    @SuppressWarnings("unchecked")
    public T makePersistent(T entity) {
        getSession().saveOrUpdate(entity);
        return entity;
    }

    public void makeTransient(T entity) {
        getSession().delete(entity);
    }

    public void flush() {
        getSession().flush();
    }

    public void clear() {
        getSession().clear();
    }

    /**
     * Use this inside subclasses as a convenience method.
     */
    @SuppressWarnings("unchecked")
    protected List<T> findByCriteria(Criterion... criterion) {
        Criteria crit = getSession().createCriteria(getPersistentClass());
        for (Criterion c : criterion) {
            crit.add(c);
        }
        return crit.list();
    }

}
```

There are some interesting things in this implementation. First, it clearly needs a Session to work, provided with setter injection. You could also use constructor injection. How you set the Session and what scope this Session has is of no concern to the actual DAO implementation. A DAO should not control transactions or the Session scope.

We need to suppress a few compile-time warnings about unchecked casts, because Hibernate's interfaces are JDK 1.4 only. What follows are the implementations of the generic CRUD operations, quite straightforward. The last method is quite nice, using another JDK 5.0 feature, *varargs*. It helps us to build Criteria queries in concrete entity DAOs. This is an example of a concrete DAO that extends the generic DAO implementation for Hibernate:

```
public class ItemDAOHibernate
        extends     GenericHibernateDAO<Item, Long>
        implements  ItemDAO {

    public Bid getMaxBid(Long itemId) {
        Query q = getSession().getNamedQuery(ItemDAO.QUERY_MAXBID);
        q.setParameter("itemid", itemId);
        return (Bid) q.uniqueResult();
    }

    public Bid getMinBid(Long itemId) {
        Query q = getSession().getNamedQuery(ItemDAO.QUERY_MINBID);
        q.setParameter("itemid", itemId);
```

```
        return (Bid) q.uniqueResult();
    }

}
```

Another example which uses the findByCriteria() method of the superclass with variable arguments:

```
public class CategoryDAOHibernate
        extends     GenericHibernateDAO<Category, Long>
        implements  CategoryDAO {

    public Collection<Category> findAll(boolean onlyRootCategories) {
        if (onlyRootCategories)
            return findByCriteria( Expression.isNull("parent") );
        else
            return findAll();
    }
}
```

Preparing DAOs with factories

We could bring it all together in a DAO factory, which not only sets the Session when a DAO is constructed but also contains nested classes to implement CRUD-only DAOs with no business-related operations:

```
public class HibernateDAOFactory extends DAOFactory {

    public ItemDAO getItemDAO() {
        return (ItemDAO)instantiateDAO(ItemDAOHibernate.class);
    }

    public CategoryDAO getCategoryDAO() {
        return (CategoryDAO)instantiateDAO(CategoryDAOHibernate.class);
    }

    public CommentDAO getCommentDAO() {
        return (CommentDAO)instantiateDAO(CommentDAOHibernate.class);
    }

    public ShipmentDAO getShipmentDAO() {
        return (ShipmentDAO)instantiateDAO(ShipmentDAOHibernate.class);
    }

    private GenericHibernateDAO instantiateDAO(Class daoClass) {
        try {
            GenericHibernateDAO dao = (GenericHibernateDAO)daoClass.newInstance();
            dao.setSession(getCurrentSession());
            return dao;
        } catch (Exception ex) {
            throw new RuntimeException("Can not instantiate DAO: " + daoClass, ex);
        }
    }

    // You could override this if you don't want HibernateUtil for lookup
    protected Session getCurrentSession() {
        return HibernateUtil.getSessionFactory().getCurrentSession();
    }

    // Inline concrete DAO implementations with no business-related data access methods.
    // If we use public static nested classes, we can centralize all of them in one source fi

    public static class CommentDAOHibernate
            extends GenericHibernateDAO<Comment, Long>
            implements CommentDAO {}

```

```
    public static class ShipmentDAOHibernate
            extends GenericHibernateDAO<Shipment, Long>
            implements ShipmentDAO {}

}
```

This concrete factory for Hibernate DAOs extends the abstract factory, which is the interface we'll use in application code:

```
public abstract class DAOFactory {

    /**
     * Creates a standalone DAOFactory that returns unmanaged DAO
     * beans for use in any environment Hibernate has been configured
     * for. Uses HibernateUtil/SessionFactory and Hibernate context
     * propagation (CurrentSessionContext), thread-bound or transaction-bound,
     * and transaction scoped.
     */
    public static final Class HIBERNATE = org.hibernate.ce.auction.dao.hibernate.HibernateDAO

    /**
     * Factory method for instantiation of concrete factories.
     */
    public static DAOFactory instance(Class factory) {
        try {
            return (DAOFactory)factory.newInstance();
        } catch (Exception ex) {
            throw new RuntimeException("Couldn't create DAOFactory: " + factory);
        }
    }

    // Add your DAO interfaces here
    public abstract ItemDAO getItemDAO();
    public abstract CategoryDAO getCategoryDAO();
    public abstract CommentDAO getCommentDAO();
    public abstract ShipmentDAO getShipmentDAO();

}
```

Note that this factory example is suitable for persistence layers which are primarily implemented with a single persistence service, such as Hibernate or EJB 3.0 persistence. If you have to mix persistence APIs, for example, Hibernate and plain JDBC, the pattern changes slightly. Keep in mind that you can also call session.connection() *inside* a Hibernate-specific DAO, or use one of the many bulk operation/SQL support options in Hibernate 3.1 to avoid plain JDBC.

Finally, this is how data access now looks like in controller/command handler code (pick whatever transaction demarcation strategy you like, the DAO code doesn't change):

```
// EJB3 CMT: @TransactionAttribute(TransactionAttributeType.REQUIRED)
public void execute() {

    // JTA: UserTransaction utx = jndiContext.lookup("UserTransaction");
    // JTA: utx.begin();

    // Plain JDBC: HibernateUtil.getCurrentSession().beginTransaction();

    DAOFactory factory = DAOFactory.instance(DAOFactory.HIBERNATE);
    ItemDAO itemDAO = factory.getItemDAO();
    UserDAO userDAO = factory.getUserDAO();

    Bid currentMaxBid = itemDAO.getMaxBid(itemId);
    Bid currentMinBid = itemDAO.getMinBid(itemId);

```

```
    Item item = itemDAO.findById(itemId, true);

    newBid = item.placeBid(userDAO.findById(userId, false),
                           bidAmount,
                           currentMaxBid,
                           currentMinBid);

    // JTA: utx.commit(); // Don't forget exception handling

    // Plain JDBC: HibernateUtil.getCurrentSession().getTransaction().commit(); // Don't forg

}
```

The database transaction, either JTA or direct JDBC, is started and committed in an interceptor that runs for every execute(), following the Open Session in View pattern. You can use AOP for this or any kind of interceptor that can be wrapped around a method call, see Session handling with AOP .

Preparing DAOs with manual dependency injection

You don't need to write the factories. You can as well just do this:

```
// EJB3 CMT: @TransactionAttribute(TransactionAttributeType.REQUIRED)
public void execute() {

    // JTA: UserTransaction utx = jndiContext.lookup("UserTransaction");
    // JTA: utx.begin();

    // Plain JDBC: HibernateUtil.getCurrentSession().beginTransaction();

    ItemDAOHibernate itemDAO = new ItemDAOHibernate();
    itemDAO.setSession(HibernateUtil.getSessionFactory().getCurrentSession());

    UserDAOHibernate userDAO = new UserDAOHibernate();
    userDAO.setSession(HibernateUtil.getSessionFactory().getCurrentSession());

    Bid currentMaxBid = itemDAO.getMaxBid(itemId);
    Bid currentMinBid = itemDAO.getMinBid(itemId);

    Item item = itemDAO.findById(itemId, true);

    newBid = item.placeBid(userDAO.findById(userId, false),
                           bidAmount,
                           currentMaxBid,
                           currentMinBid);

    // JTA: utx.commit(); // Don't forget exception handling

    // Plain JDBC: HibernateUtil.getCurrentSession().getTransaction().commit(); // Don't forg

}
```

The disadvantage here is that the implementation classes (i.e. ItemDAOHibernate and UserDAOHibernate) of the persistence layer are exposed to the client, the controller. Also, constructor injection of the current Session might be more appropriate.

Preparing DAOs with lookup

Alternatively, call HibernateUtil.getSessionFactory().getCurrentSession() as a fallback, if the client didn't provide a Session when the DAO was constructed:

```
public abstract class GenericHibernateDAO<T, ID extends Serializable>
        implements GenericDAO<T, ID> {

```

```
    private Class<T> persistentClass;
    private Session session;

    public GenericHibernateDAO() {
        this.persistentClass = (Class<T>) ((ParameterizedType) getClass()
                                .getGenericSuperclass()).getActualTypeArguments()[0];
    }

    public void setSession(Session session) {
         this.session = session;
    }

    protected void getSession() {
        if (session == null)
            session = HibernateUtil.getSessionFactory().getCurrentSession();
        return session;
    }

...
```

The controller now uses these stateless data access objects through direct instantiation:

```
// EJB3 CMT: @TransactionAttribute(TransactionAttributeType.REQUIRED)
public void execute() {

    // JTA: UserTransaction utx = jndiContext.lookup("UserTransaction");
    // JTA: utx.begin();

    // Plain JDBC: HibernateUtil.getCurrentSession().beginTransaction();

    ItemDAO itemDAO = new ItemDAOHibernate();
    UserDAO userDAO = new UserDAOHibernate();

    Bid currentMaxBid = itemDAO.getMaxBid(itemId);
    Bid currentMinBid = itemDAO.getMinBid(itemId);

    Item item = itemDAO.findById(itemId, true);

    newBid = item.placeBid(userDAO.findById(userId, false),
                           bidAmount,
                           currentMaxBid,
                           currentMinBid);

    // JTA: utx.commit(); // Don't forget exception handling

    // Plain JDBC: HibernateUtil.getCurrentSession().getTransaction().commit(); // Don't forg

}
```

The only disadvantage of this very simple strategy is that the implementation classes (i.e. ItemDAOHibernate and UserDAOHibernate) of the persistence layer are again exposed to the client, the controller. You can still supply a custom Session if needed (integration test, etc).

Each of these methods (factories, manual injection, lookup) for setting the current Session and creating a DAO instance has advantages and drawbacks, use whatever you feel most comfortable with.
Naturally, the cleanest way is managed components and EJB 3.0 session beans:

Writing DAOs as managed EJB 3.0 components

Turn your DAO superclass into a base class for stateless session beans (all your concrete DAOs are then stateless EJBs, they already have a business interface). This is basically a single annotation which you

could even move into an XML deployment descriptor if you like. You can then use dependency injection and get the "current" persistence context provided by the container:

```
@Stateless
public abstract class GenericHibernateDAO<T, ID extends Serializable>
        implements GenericDAO<T, ID> {

    private Class<T> persistentClass;

    @PersistenceContext
    private EntityManager em;

    public GenericHibernateDAO() {
       setSession( (Session)em.getDelegate() );
    }

...
```

You can then cast the delegate of an EntityManager to a Hibernate Session.
This only works if you use Hibernate as a Java Persistence provider, because the delegate is the Session API. In JBoss AS you could even get a Session injected directly. If you use a different Java Persistence provider, rely on the EntityManager API instead of Session. Now wire your DAOs into the controller, which is also a managed component:

```
@Stateless
public class ManageAuctionController implements ManageAuction {

    @EJB ItemDAO itemDAO;
    @EJB UserDAO userDAO;

    @TransactionAttribute(TransactionAttributeType.REQUIRED) // This is even the default
    public void execute() {

        Bid currentMaxBid = itemDAO.getMaxBid(itemId);
        Bid currentMinBid = itemDAO.getMinBid(itemId);

        Item item = itemDAO.findById(itemId, true);

        newBid = item.placeBid(userDAO.findById(userId, false),
                               bidAmount,
                               currentMaxBid,
                               currentMinBid);

    }
}
```

P.S. Credit has to be given to Eric Burke, who first posted the basics for this pattern on his blog. Unfortunately, not even the Google cache is available anymore.

A Better typed Generic DAO? You decide!

We are missing something on <T> end, since T allows you to "domain-ify" everything! and the Identifier type should ideally match the identifier type of T, but there's no way to do that on the code above. You decide which approach is better.

```
// Our common Model interface that an abstract Domain model will implement and all domain //
public interface IModel<ID extends Serializable> {
        public abstract ID getId();
        public abstract void setId(final ID pId);
}
// Our generic DAO, NOTE: MODEL's ID type is the same as ID now, which makes sense.
// Also model type is more restrictive, dis-allowing all kinds of funky stuff to go in.
public abstract class GenericHibernateDAO<MODEL extends IModel<ID>, ID extends Serializable>
```

```
09.
10.     private Class<MODEL> persistentClass;
11.     private Session session;
12.
13.     public GenericHibernateDAO() {
14.         // FIXME : I don't like magic number in the code, is there any way to fix 0 to someth
15.         this.persistentClass = (Class<MODEL>) ((ParameterizedType) getClass()
16.                                 .getGenericSuperclass()).getActualTypeArguments()[0];
17.     }
18.
19.     public final void setSession(final Session pSession) {
20.         this.session = session;
21.     }
22.
23.     protected void getSession() {
24.         if (session == null)
25.             session = HibernateUtil.getSessionFactory().getCurrentSession();
26.         return session;
27.     }
28. ...
```

In addition, we could add things like:

```
01. public final String getRootAlias(){
02.  this.getPersistentClass().getSimpleName() + String.valueOf('_');
03. }
```

Alhough this is not necessary or part of the enhanced version, but when criteria API is in use, this comes in handy.

　　分类：　　标签： example, dao

6 评论

null null 2010-3-20 上午11:41

I am not sure I even see a use for DAO's in EJB3. Seems like a lot of extra classes and code for not much benefit. Recently, we have just been putting findXXXById or findAllActiveXXX in the bean itself where the return value of the find method matches the bean. Each bean typically ends upw with an average of 1 method with some having 4 or 5 methods and others having 0, but the average being 0. Since everything is a pojo that is returned, I just don't see the benefit of all this extra code at all. Maybe I am missing something?

Take your execute method above with 6 lines of code. Without the DAO's and just using Item.findByXXX(), it is 4 lines of code instead of 6. It turns out there is about a 40% reduction in code overall when we looked at it. Why have 40% more code to have DAO's...what is it buying for all that extra typing. I just don't get it personally.

操作 喜欢 (0)

Frederico Pereira 2010-6-28 下午6:57

Hi, thanks for the article. I have a question about using JTA for transaction demarcation. Do I have always to start a transaction (utx.begin()) even for a simple 'query' dao operation (like findById(), for instance)? I´m asking this because I´m getting an exception when using sessionFactory.getCurrentSession() and not having started a UserTransaction before. It seems that a JTA transaction must be started before using getCurrentSession() method because of persistence context propagation uses the JTA context and not the thread local context. Is that correct? Thanks.

操作 喜欢 (0)

Ilya Sorokoumov 2011-1-11 上午9:46 (回复 Frederico Pereira)

I wrote a post in my blog on the similar topic. I guess that we still need an analogue of DAO in JPA but I'm not sure that we should name it DAO. =)
http://community.jboss.org/people/ilya40umov/blog/2011/01/06/genericejb-based-jpa-entitymanager-extension

操作 喜欢 (0)

Michal Pp 2011-6-23 下午12:24

The findByExample() method has different signature on the interface and on the implementation.

操作 喜欢 (0)

Kuldeep Mahajan 2013-3-28 下午1:10

Hi, thanks for this article.

I just have one suggestion for the getSession method signature error.

It should return Session object insted of void.

Following are the classes which contain getSession method:

```
Class: IModel and GenericHibernateDAO
```

操作 喜欢 (0)

Ammar Ali 2014-1-27 上午7:21

Hello,

Is there implemented example available ?

thanks,

操作

喜欢 (0)