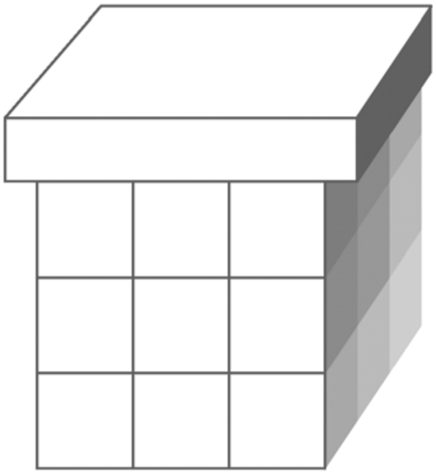
CHAPTER 3

TRANSACTIONS



OBJECTIVES

After completing "Transactions," you will be able to:

- **Describe the Spring approach to unifying transaction control.**
- **Implement declarative transactions for a Spring Web application.**

Transaction Control in Java EE

- **Traditionally, Java developers have been faced with some unpleasant choices when it comes time to implement transaction control for their applications.**
  - Direct transaction management via **JDBC** is clunky, nearly unmaintainable, and can't support multiple databases effectively.
  - The **Java Transactions API**, or **JTA**, offers support for distributed transactions (i.e. multiple databases), but is even less appealing as an API and carries considerable weight, including a semi-dependency on JNDI as a way of publishing and sharing transaction contexts.
- **Java EE's primary solution has been container-managed transactions in EJB containers, most recently according to the Java Persistence API specification.**
  - This is essentially a JTA solution, but CMT hides the JTA details.
  - What's good about CMT is that it recognizes that **transactionality is a cross-cutting concern**, a general feature common to otherwise very different business objects.
  - In fact much of EJB's **deployment descriptor** vocabulary can be seen as an early attempt at **aspect-oriented programming**: declarative development in general is about identifying generic features that can be implemented on behalf of a class, rather than the class having to seek out and call a reusable utility on its own.
  - What's not so good about CMT is that it requires an EJB container, and hence a full-blown Java EE application server.

Spring's Transaction Management

- **Spring takes the basic concept of declarative transactions and runs away with it – it leaves the weight of EJB behind.**
  - It's Spring's philosophy all over again, that ordinary Java beans should enjoy the declarative instantiation, management, services, lifecycle, and context traditionally reserved for specialized stereotypes such as servlets and EJBs.
  - Spring looks at declarative transactions and asks, "Why can't ordinary Java classes have those?"
- **The building blocks of transaction management in Spring are:**
  - A **transaction manager**, of which Spring offers many off-the-shelf implementations
  - A **data source** or other transactional resource
  - **Transaction advice** attached to one or more transactional resources, spelling out transaction requirements and attributes
- **These can all be supplied declaratively or programmatically.**
  - A completely declarative approach is most common.
- **Spring's transaction model does not require the services of a heavyweight container, as EJB's model does.**
- **However, it cannot support distributed transactions.**
  - This compromise is acceptable for most applications, which will only use a single database anyway.
  - Or, an application might use several databases, but never need to coordinate transactions over more than one at a time; that too is workable with Spring transaction management.

Declarative Transactions

- **Steps to implementing transactional behavior for your Spring application are as follows.**
  - We'll speak the language of JDBC for the moment; but all the following concepts map one-for-one to other persistence techniques, including O/R mapping tools and JPA proper.

1. Declare your data source as a global bean – we've seen an example of this already.
2. Declare a transaction manager and inject the data source into it.

```
<bean
  id="transactionManager"
  class="org.springframework.jdbc.datasource
                .DataSourceTransactionManager"
>
  <property name="dataSource" ref="dataSource" />
</bean>
```

3. For each transactional class or method, declare transactional advice and attach that advice to the class or method.

- **There are several alternatives for this last piece.**
- **This is the aspect-oriented part of the puzzle, and AOP practice for Java is evolving rapidly.**
  - The longest-standing approach uses **Spring AOP**.
  - If coding in Java 5, you can use **annotations**.
  - There is also support for **AspectJ**.

Spring AOP vs. Annotations

- **Spring AOP and Tomcat don't mix so well.**
  - The CGLIB code generator eventually drains the "PermGen" memory space that a Java VM expects to be reserved for ordinary, i.e. static, class definitions.
  - Successive redeployments to Tomcat will continually re-generate AOP classes, and gradually the heap space will be drained.
  - This is a non-starter for most professional situations.
- **AspectJ doesn't suffer this embarrassment, but it's not as naturally bound to Spring, either.**
- **The Java-5.0 annotation is actually the new kid on the block, but it's a proven, portable, and well-tested feature of the language.**
  - The downside is that annotations live in Java source files, and so **can't be modified independently** of the class.
  - Generally this is a significant caution regarding the use of annotations, but they have their place, and transactions are a good example of appropriate use.
  - **Transaction attributes**, once declared, are **not usually volatile**, or if they are changing it's usually in concert with significant code changes anyway.
  - Spring's **@Transactional** annotation is dead-simple to use, and does have the advantage of making it easy to see in source code and javadoc what interfaces, classes, and methods offer which transactional characteristics.

The @Transactional Annotation

- **org.springframework.transaction.annotation.Transactional** can be applied to classes, interfaces, and methods.

```
public interface Transactional
  extends Annotation
{
  public Propagation propagation ();
  public Isolation isolation ();
  public int timeout ();
  public boolean readOnly ();
  public Class[] rollbackFor ();
  public String[] rollbackForClassName ();
  public Class[] noRollbackFor ();
  public String[] noRollbackForClassName ();
}
```

  - As you can see, it can support precise definitions of transactional behavior, including isolation level and propagation characteristics (e.g. what happens if a transaction is already in force).
  - The **rollbackFor** array allows you to declare what sorts of exceptions should trigger rollbacks. The default is to roll back on any runtime exception.

- **Spring does bend the rules a bit regarding inheritability with this annotation.**
  - It is **@Inherited**, but Spring also respects inheritance of transactionality under interfaces, and for individual methods; this is inconsistent with standard annotation processing.
  - This is a minor wrinkle, but it can confuse other annotation processors such as documentation generators.

Enforcing Transactions

DEMO

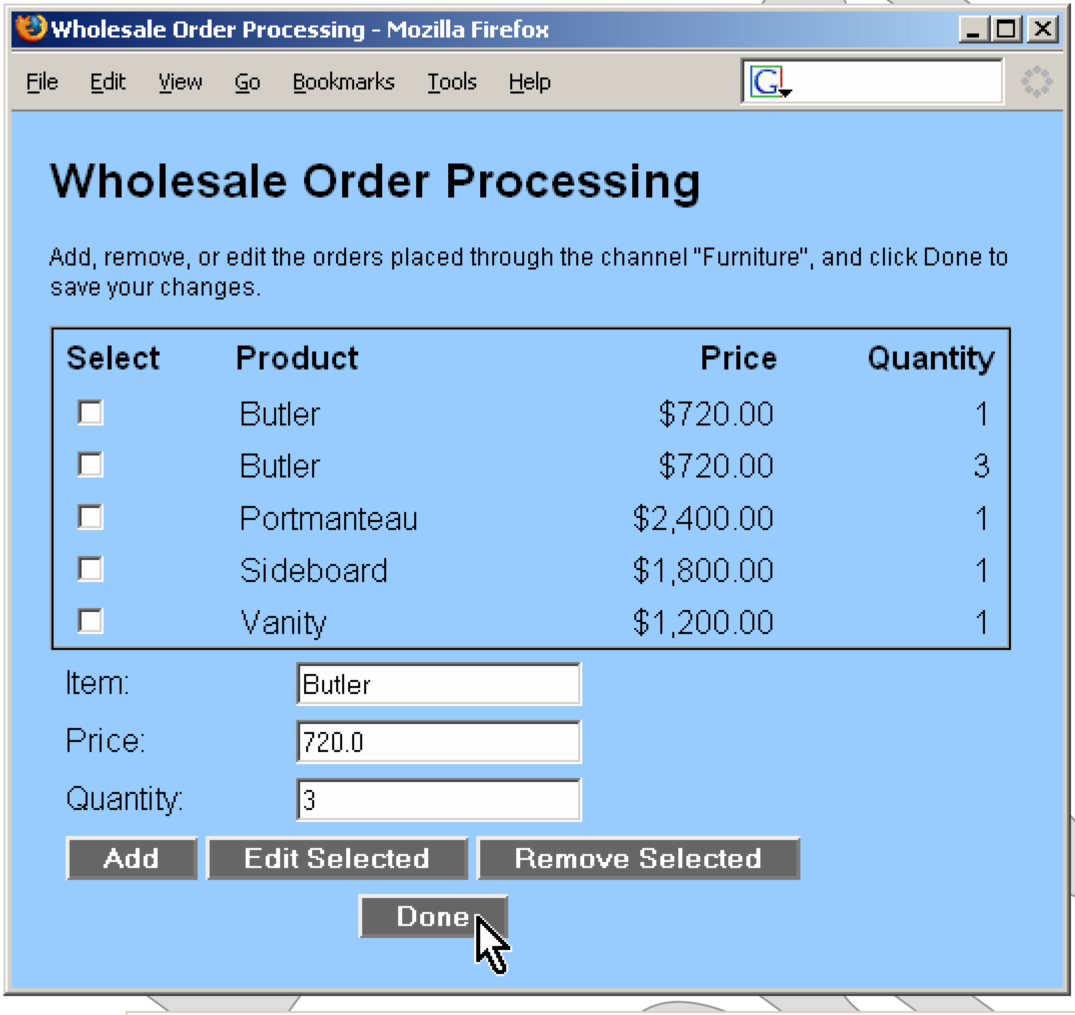
- **We'll conclude our study of persistence tiers in Spring applications by adding transaction support to the Wholesale application.**
  - Do your work in **Demos/Transactions**.
  - The completed demo is in **Examples/Wholesale/Step3**.

1. First, test the transactional behavior of the starter application. You may have noticed that the database includes an overlaid uniqueness constraint for the **ordr** table: duplicate listings of a given product are not allowed within a feed. That is, no two rows can have the same feed name and product name.

Enforcing Transactions

DEMO

Build and deploy; edit one of the feeds by changing the second item to have the same product name as the first, then click **Done**:



Wholesale Order Processing

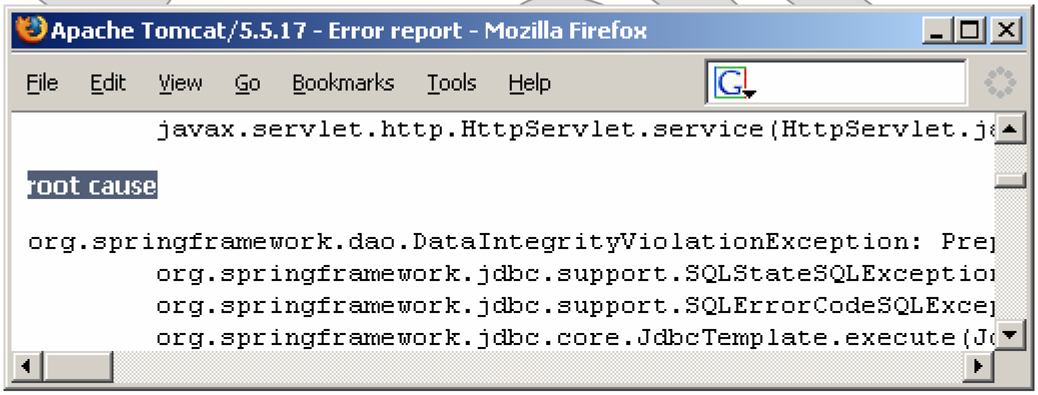
Add, remove, or edit the orders placed through the channel "Furniture", and click Done to save your changes.

Select	Product	Price	Quantity
☐	Butler	$720.00	1
☐	Butler	$720.00	3
☐	Portmanteau	$2,400.00	1
☐	Sideboard	$1,800.00	1
☐	Vanity	$1,200.00	1

Item: Butler

Price: 720.0

Quantity: 3



```
        javax.servlet.http.HttpServlet.service(HttpServlet.j

root cause

org.springframework.dao.DataIntegrityViolationException: Pre
        org.springframework.jdbc.support.SQLStateSQLExceptio
        org.springframework.jdbc.support.SQLErrorCodeSQLExce
        org.springframework.jdbc.core.JdbcTemplate.execute(J
```

Enforcing Transactions

DEMO

2. You get an error as you run afoul of this constraint. But what's more interesting is what's happened to the data for that feed. Go to **feeds.jsp** again, and edit the same feed. See that all the feed contents except for the first order are now gone. They missed the boat! When the error occurred, the process was simply terminated, with one order in, a bad one rejected, and the rest of the order simply vanished.

3. Open **docroot/WEB-INF/Database.xml**, and notice that a second XML namespace is now supported for this document. This is the Spring transactions schema, with the namespace URI:

```
http://www.springframework.org/schema/tx
```

4. Declare a transaction manager for the web application – the autowiring will seek out and connect to the data source that's already declared:

```
<bean
  id="transactionManager"
  class="org.springframework.jdbc.datasource
                .DataSourceTransactionManager"
  autowire="byType"
/>
```

5. And now for the magic words: "annotation-driven." Declare this one feature as follows, and it enables Java-5.0 transaction annotations throughout the application:

```
<tx:annotation-driven />
```

Enforcing Transactions

DEMO

6. Now, in **src/cc/sales/OrderDAO.java**, import the **@Transactional** annotation:

```
import org.springframework.transaction
                        .annotation.Transactional;
```

7. ... and declare it for all three methods, as in:

```
@Transactional
public void save
    (List<Order> feed, String feedName)
  throws Exception;
```

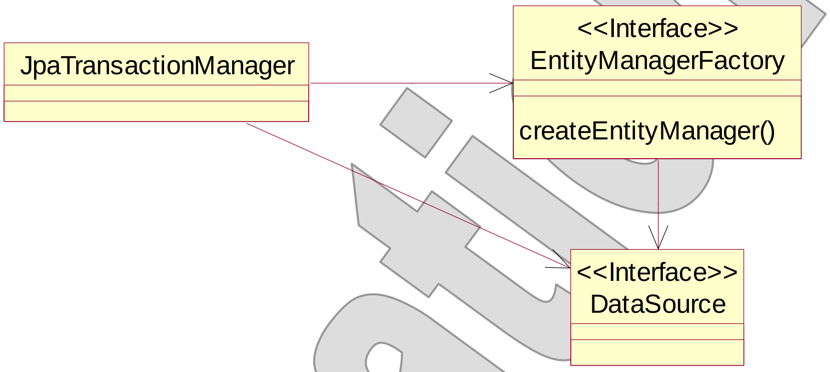
8. Build the application and re-populate the database:

```
ant
run PrimeWithData
```

9. Test the same use case, creating a duplicate product in one of the existing feeds. You'll get the same error page back, but when you visit that feed again, you'll see clearly that the transaction was rolled back: none of your edits were committed to the database, and so not only integrity but consistency has been preserved.

The JpaTransactionManager

- **Spring provides a transaction manager for JPA operations as well:**



  - Ultimately it governs transactions over database connections.
  - But as a matter of configuration it can usually just be attached to an **EntityManagerFactory**.
  - It will query that object for its underlying **DataSource**, and then monitor connections from there.

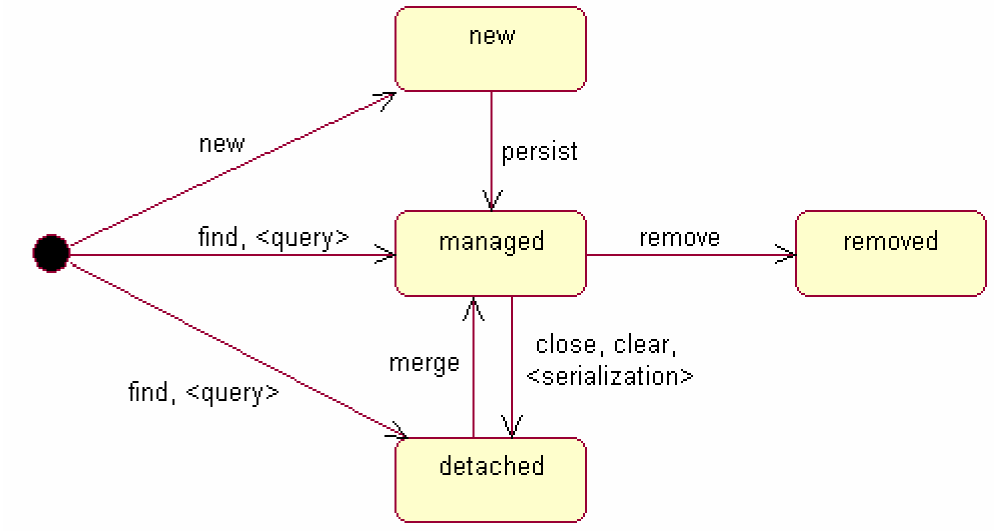
- **This solution snaps into JPA applications just as easily as the DataSourceTransactionManager does for JDBC applications.**

Transactions in JPA

- **The impact of transactions on JPA code is a little different though.**
- **Or, better to say that the impact of not having defined transactions is different.**
- **JDBC code will execute in the absence of transactions, with effects that will be determined by vendor-specific settings such as auto-commit.**
- **JPA will not carry out any updates without transaction in force.**
  - **EntityManager** operations that result in changes – **persist**, **merge**, **remove**, etc. – will throw the **TransactionRequiredException** if a transaction is not in force.
  - What happens from there depends on who's invoking the method and how they handle the exception.
  - For better or worse, Spring's interceptors actually gobble up this exception in most cases, resulting in quiet failure rather than a crash.
- **This is why we've not yet seen any capability in the LandUse application to write data, even when the user executes Add, Remove, Done, and so forth.**
- **We're about to fix that!**

Entity States

- **JPA entity instances – as Java objects in memory – can be in one of three states: new, managed, detached, and removed.**



  - The managed state is sometimes called the persistent state.
  - Only entities in the persistent state can be saved or removed.

- **As we've said, an entity manager will only carry out write operations within a transaction.**
- **But transactions matter to read operations, too – though the effects are more subtle.**
- **When reading the database, JPA operations will return entities that are in one of two states:**
  - The managed state, if a transaction was in force
  - The detached state, otherwise
- **So the lack of a transaction in one method can result in a failure in another method, by setting the stage for an attempt to write on an object that is not in the managed state.**

Extended Persistence Contexts

- One JPA feature, and something that will come up in the lab exercise in a moment, is the **extended persistence context**.
- There is an attribute to **@PersistenceContext** that dictates how context boundaries should be set – with possible values:

```
PersistenceContextType.TRANSACTION
PersistenceContextType.EXTENDED
```

- So far, we've been working with persistence contexts whose context type has been **TRANSACTION**.
  - This means that a persistence context has been created for a given transactions, and **closed or cleared** when that transaction has committed or rolled back.
  - Our transactions have been wrapped around individual method calls, so this has amounted to a **method-scoped context**.
  - When a context closes, all entities in that context become **detached**.
- This works fine in most cases, but when a service or DAO is stateful and needs to hang on to entities in a persistent state from one method call to the next, we need a different solution.
- One way would be a long-running transaction, but that's expensive and can create major concurrency headaches.
- So JPA offers the **EXTENDED** context, which lives for as long as the target object lives – that is, we get a bean-scoped context.
  - This allows entities once derived to stay "in context" and stay in the managed state, for use in later method calls.

LandUse Transaction Advice

LAB 3

Suggested time: 45 minutes

In this lab you will complete the LandUse application by configuring a transaction manager and declaring transaction advice for individual DAO methods.

Detailed instructions are found at the end of the chapter.

SUMMARY

- **Spring's transaction support strikes a nice balance between feature set and simplicity.**
  - The major simplifying assumption is that there's a single database.
  - Thus the lightweight Spring container can support declarative transactions – just like the big boys! – and with less work.
- **While Spring brings different improvements to JDBC versus JPA (or proprietary ORM) coding, the value-add of transaction control via Spring is more consistent, regardless.**
  - Neither JDBC nor JPA defines a built-in transaction manager.
  - They both rely on outside agents for transaction control, and they both have APIs for engaging with those outside agents.
  - Spring can be that agent, replacing an EE application server in this function.
  - Spring also makes transactions – and DAO support in general – available to any Java class, and not just to a select few types of managed object as in Java EE.

LandUse Transaction Advice

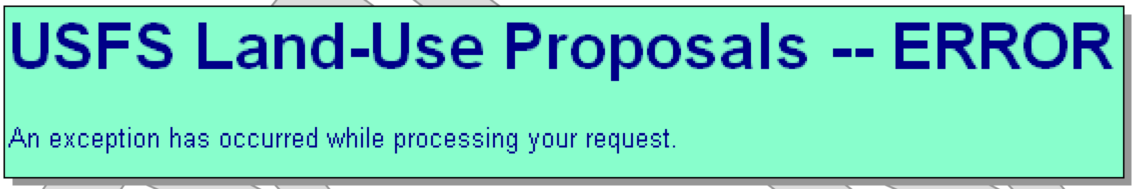
LAB 3

In this lab you will complete the LandUse application by configuring a transaction manager and declaring transaction advice for individual DAO methods.

Lab workspace:	**Labs/Lab3**
Backup of starter code:	**Examples/LandUse/Step2**
Answer folder(s):	**Examples/LandUse/Step3**
Files:	**docroot/WEB-INF/LandUse-servlet.xml** **src/gov/usda/usfs/landuse/jpa/ProposalServiceImpl.java** **src/gov/usda/usfs/landuse/jpa/LandUseServiceImpl.java**

Instructions:

1. Build and test the starter version of the application; this is exactly as we left it at the end of the demonstration in the previous chapter. Try a couple of use cases in particular – browser screenshots and snippets from the server console are shown for each:
   - Choose a proposal and click **Remove**.

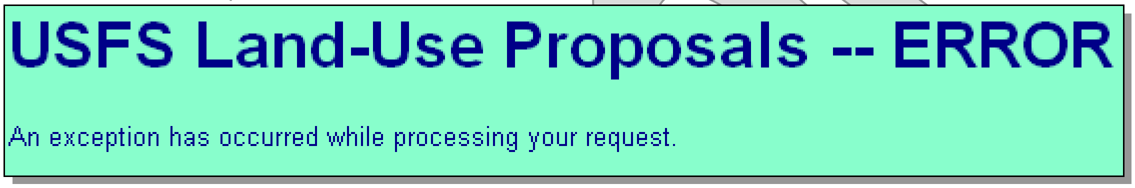


USFS Land-Use Proposals -- ERROR

An exception has occurred while processing your request.

```
SEVERE: Exception in request processing.
java.lang.IllegalArgumentException: Entity must be managed to call
remove: (1,Green Mountain NF,Ski USA,Alpine park), try merging the
detached and try the remove again.
```

   - Choose a proposal and click **Edit**. Then, make a change to one of the fields and click **Done**.



USFS Land-Use Proposals -- ERROR

An exception has occurred while processing your request.

```
oracle.toplink.essentials.exceptions.ValidationException
Exception Description: Cannot persist detached object ...
```

Both of these failures result from a lack of transaction control.

2. Open **LandUse-servlet.xml** and add a transaction manager to the application context:

```
<bean
  id="transactionManager"
  class="org.springframework.orm.jpa.JpaTransactionManager"
  autowire="byType"
/>
```

   The effect of autowiring here is to connect the transaction manager to the entity manager factory that's already in place.

3. Add the declaration that lets Spring know that we'll attach transaction advice to classes and methods using Java annotations:

```
<tx:annotation-driven />
```

4. Build and test the **Remove** button again … no joy, and of course we wouldn't expect much change yet, since we've not actually given any transaction advice.
5. Open **LandUseServiceImpl.java**, and add the **@Transactional** annotation to the **withdraw** method.

```
@Transactional
public void withdraw (int ID)
```

6. Build and test again, and you should see that **Remove** works now.
7. Open **ProposalServiceImpl.java**, and mark **update** as **@Transactional**.
8. Build and test the **Edit/Done** use case – hmm, still no dice. **update** calls **em.persist** in an attempt to save changes to the **model** object. This call does require a transaction, and you're now having Spring provide one. But it also requires a managed object, and **model** is not currently in the managed state.

   Why not? Where did we get the **model**, and how do we know it is or isn't in a managed state? It was derived in an earlier call to **init** – and, no, that method's not transactional. So we have a detached object, and hence the failure when we try to save changes.

9. Mark **init** as being **@Transactional** as well, and test again. No? Still not working?

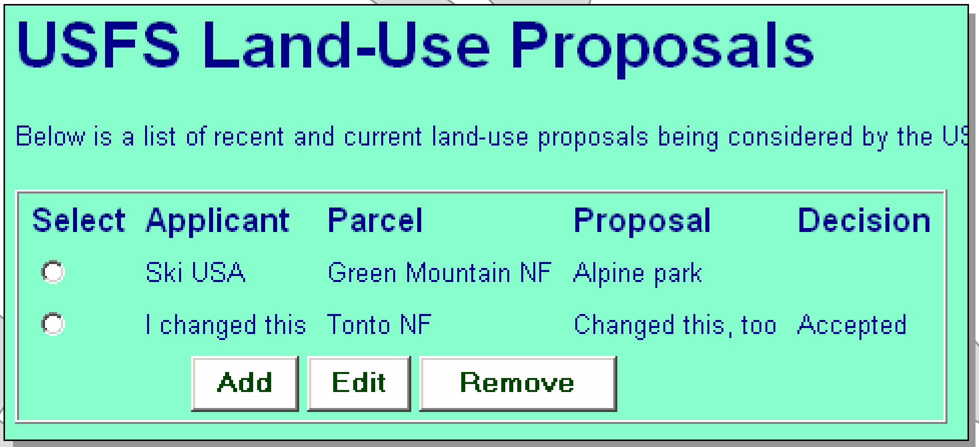
   One last mystery to solve here, and this is where the extended persistence context comes into play. It's fine (and necessary) to mark both **init** and **update** as transactional: the first one must initialize **model** to an entity in the managed state, and the second must write that object, so they each need a transaction. The remaining problem is not really about transaction boundaries, but context boundaries: the **model** is initially managed, but is detached from the context after **init** returns.

10. Solve this by making the context used by **ProposalServiceImpl** extended:

```
@PersistenceContext (type=PersistenceContextType.EXTENDED)
private EntityManager em;
```

   Import **PersistenceContextType** from **javax.persistence**.

11. Build and test. Now you see that you can save changes through the **Done** command.



USFS Land-Use Proposals

Below is a list of recent and current land-use proposals being considered by the U

Select	Applicant	Parcel	Proposal	Decision
○	Ski USA	Green Mountain NF	Alpine park	
○	I changed this	Tonto NF	Changed this, too	Accepted

12. You can carry this exercise forward as far as you like, and gradually bring the editing features of the application online. The remaining methods that will need to be transactional are:

    - On **LandUseServiceImpl**, methods **getAll**, **getProposal**, and **submit**
    - On **ProposalServiceImpl**, methods **decide**, **addPublicComment**, and **addProfessionalComment**