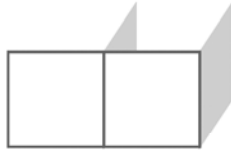
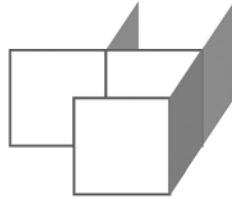




CHAPTER 3
SESSION BEANS



OBJECTIVES

After completing "Session Beans," you will be able to:

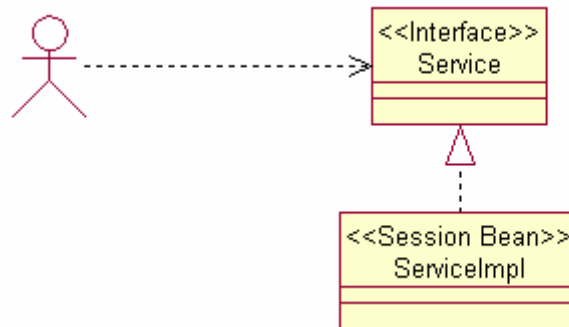
- Explain the functioning of stateful and stateless session beans.
- Describe the lifecycle and context interactions between container and session bean.
- Implement stateless session beans as interfaces to enterprise business logic.
- Understand the impact of the stateful/stateless choice on client usability and server-side performance.
- Implement stateful session beans to support multiple-request use cases for clients.

The Mission

- The session bean provides an interface to business functionality.
- It acts as the entry point to an EJB module – often working with entities to carry out a requested task.
- A few perspectives on session beans are illustrative:
 - Session beans exemplify the classic **Façade** design pattern, by which an object offers a simplified interface to a more intricate system, hiding the internals of that system and making the client's interactions easier (and the system itself more secure).
 - The industry has seen a trend toward **service-oriented architecture**, or **SOA**, which organizes functionality into **coarse-grained, stateless, loosely-coupled** services. Session beans (especially stateless ones) can implement an SOA.
 - As mentioned in the previous chapter, the EJB Core specification refers to session beans as “logical **extensions of the client** program that run on the server.”
 - This is an interesting perspective indeed, suggesting that the session bean acts as an agent of the client, more than as a representative of the server side of the application.
 - Whichever way we look at it, session beans are certainly designed with **specific client use cases** in mind – to facilitate specific, anticipated interactions and scenarios.

Interface/Implementation Split

- As service objects, session beans typically observe an **interface/implementation split**:



- **The interface defines a contract between the client and the EJB.**
 - It is more formally known as the **business interface**.
 - It is defined as a simple, classic Java **interface**.
 - The container uses the business interface to generate stubs, proxies, and other plumbing around the client-EJB interaction.
- **The session bean implements one or more business interfaces.**
 - It is hidden from the client; it interacts only with the container.
 - The session bean is a normal Java class, with metadata that announces it as a session bean to the EJB container.
 - This metadata can be as simple as the **@Stateless** annotation on the class, as complex as a full-blown, EJB-2.1-style deployment descriptor, or anything in-between.

Stateful vs. Stateless

- EJB identifies two subclasses of session beans.
 - **Stateful** beans hold **conversational state** – that is, information specific to an ongoing conversation with a specific client.
 - **Stateless** beans are just that; any number of clients can call them and every call is a fresh request.
- **This choice presents a trade-off of usability for scalability.**
 - Stateful beans can be easier for clients to use, especially where there are multiple requests in a typical use case and things for the bean to remember about previous requests.
 - For such a use case, a stateless bean requires that all information necessary to carrying out a given request be provided with that request – it has no memory.
 - There is a cost for this usability, because when handling a high volume of stateful requests the container is challenged to keep a lid on the number of beans in memory.
 - Otherwise the application's performance will scale poorly as the memory usage gets out of control.
- **We'll come back to the mechanics of stateful and stateless beans later in the chapter.**

The @Stateless Annotation

- Identify a Java class as a stateless session bean with the annotation **@javax.ejb.Stateless**.

```
@Stateless
public class MySessionBeanImpl
    implements MySessionBean
{ ... }
```

- Often, it's as simple as that!
 - If you want the bean to be available outside the enterprise application (outside the EAR file), add the **@Remote** annotation:

```
@Stateless @Remote
public class ...
```

- The annotation does define several optional attributes:
 - **name** gives the resulting bean a specific name, which can be used to identify it from other beans and clients. The default bean name is the simple name of the class itself (no package tokens).

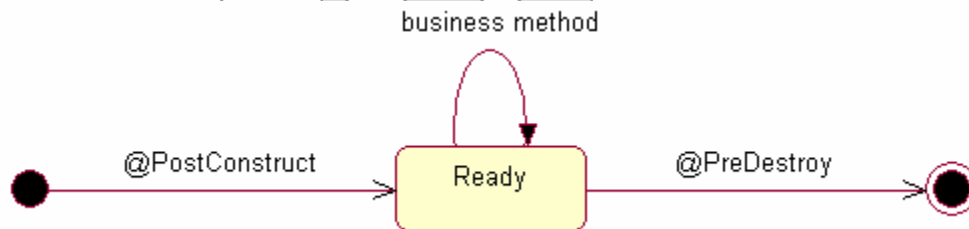
```
@Stateless (name="NotMySessionBeanImpl")
public class MySessionBeanImpl ...
```

- **mappedName** defines a global JNDI name for the bean. This is only meaningful for a remote bean and it is non-portable – different application servers will handle JNDI differently.

```
@Stateless (mappedName="ejb/GreatService")
public class MySessionBeanImpl ...
```

Lifecycle and State Transitions

- A stateless session bean has a very simple lifecycle.
- Typically – and this is not mandated by the specification, but it's common practice:
 - The container will create a single instance of the bean when the application is deployed, the server is started up, or on the first request to the bean.
 - It will destroy the bean on undeploy or server shutdown.
- The actual container behavior may vary – there may even be a pool of objects – but the basic contract is the same:



- The stateless bean is created and is immediately ready to process requests.
- It is destroyed when no longer needed.

Lifecycle Hooks

- The session bean may need to perform additional tasks at various points in its lifecycle.
- EJB 2.1 defined a required lifecycle interface for each bean type.
 - The bean had to implement this interface, which was full of **lifecycle hook methods** such as **ejbCreate** and **ejbRemove**.
 - Many of these were usually empty methods, and the interface model was poorly factored, so for example a stateless session bean still had to provide meaningless implementations of **ejbPassivate** and **ejbActivate**.
- EJB 3.0 uses annotations to identify any lifecycle hooks.
- For stateless beans there are just two interesting options – these annotations are defined in package **javax.annotation**:
 - Use **@PostConstruct** to identify a method that should be called to initialize a newly-created bean:
`@PostConstruct public void anyMethod () { ... }`
 - Use **@PreDestroy** for a method that performs clean-up tasks:
`@PreDestroy public void anyMethod () { ... }`
 - The method signature must be as shown above; the method(s) can have any visibility but cannot be **static** or **final**.

Session Context

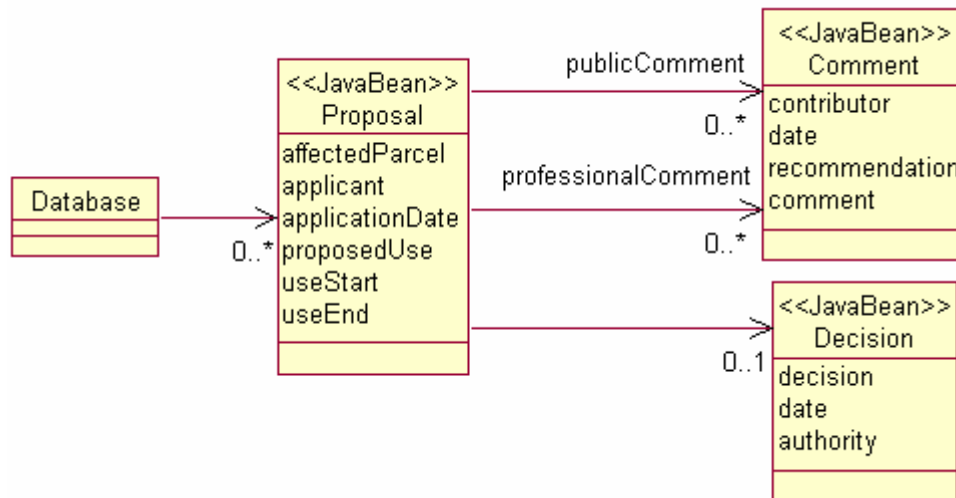
- In the previous chapter we spoke of lifecycle and context interfaces as the two channels of communication between the bean and the container.
- Lifecycle hook methods implement one channel, allowing the container to call the bean.
- The bean can call the container through the **SessionContext** interface.
- An object implementing this interface will be injected into a field or setter method with the **@Resource** annotation whose type is **SessionContext**:

```
@Resource private SessionContext context;
```

- The session context offers many features, and most are beyond the scope of this course.
- Some particularly useful methods are:
 - **lookup**, which takes a simple name to be found in the session bean's **component environment** – we'll delve into this in detail in a later chapter
 - **isCallerInRole**, which gives the bean a way to perform authorization at a finer grain than the container can provide
 - Various methods pertaining to transaction control

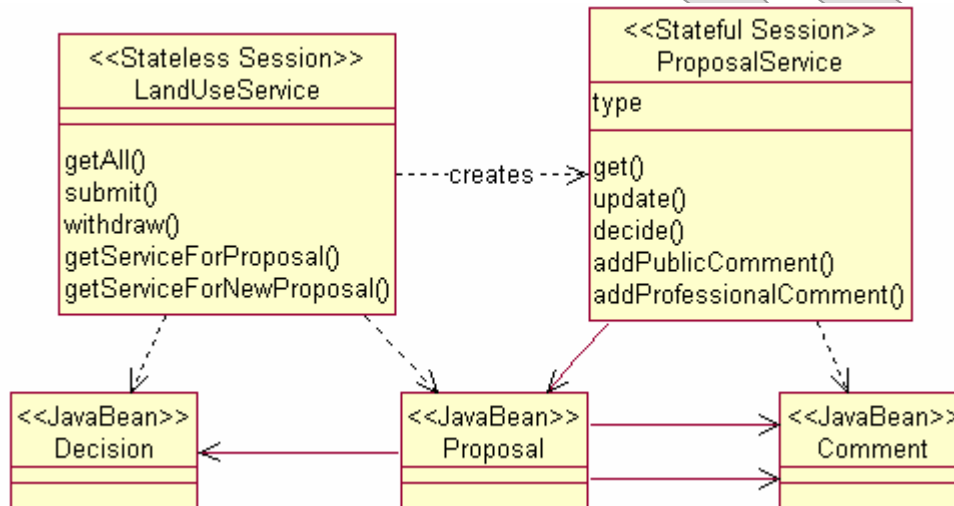
The LandUse Case Study

- We'll now take the first few steps in building the primary case study for the course: an application that presents a database of proposals for use of public lands, and allows editing and final decision-making on the proposals.
- The domain model is shown in summary here:



The LandUse Case Study

- Over two labs in this chapter, we'll add a service layer comprising one stateful bean and one stateless bean:



- A web interface of servlets and JSPs is already in place, with code that will call these session beans commented out for the moment.
- In later chapters we'll convert the domain model from POJOs to proper JPA entities and connect to a relational database.
 - For the moment we load the data from a single serialized-object file, just to give us something to work with.

The LandUseService

LAB 3A

Suggested time: 30-45 minutes

In this lab you will implement the **LandUseService** stateless session bean. This will enable part of the web interface – by the end of the lab the user will be able to view a summary of all proposals, navigate to a non-editable detail view of any one proposal, and add and remove proposals. Other functions will come online in a later lab when you build the **ProposalService**.

Detailed instructions are found at the end of the chapter.

Evaluation Only

Pooling Stateful Objects

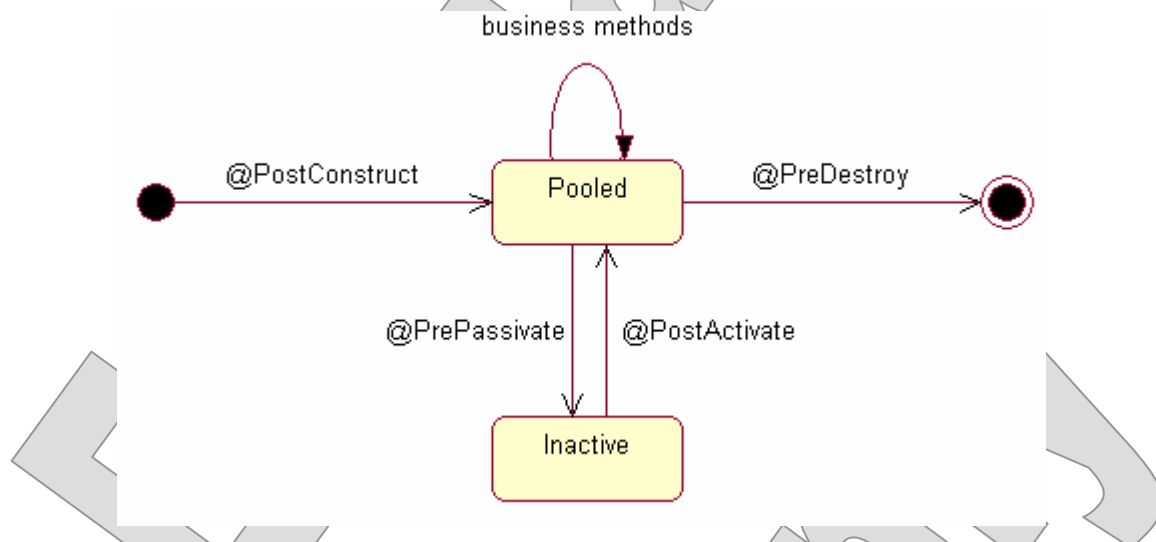
- The EJB container must handle a potentially high volume of client requests to session beans in an efficient way.
- First off, it will typically pool request-handling threads.
- Then, it will manage creation of beans differently based on the statefulness of the bean class.
- It may instantiate a stateless bean once – it is a **singleton** for all practical purposes – or it may create a pool of identical objects.
 - There's no need for more objects; potentially hundreds of request threads can run freely over the bean code, because the bean has no state for those threads to share and possibly corrupt.
- But the container has no choice but to create multiple stateful beans.
 - It must control the total number of objects in memory, so it will typically implement a **pool** of recyclable objects.
 - To recycle a pooled, stateful bean, the container must store off the bean's conversational state; re-load any state for the client making the current request into the bean, and then call the business method.
 - This is a process known as **passivation and activation**, and it amounts to automatic Java serialization and de-serialization of non-transient fields defined on the session bean.
 - It's the best the container can do, but still less efficient than the use of a singleton for stateless beans.

The @Stateful Annotation

- Inform the container that it's dealing with a stateful bean using the **@Stateful** annotation:

```
@Stateful
public class MyStatefulBeanImpl
    implements MyStatefulBean
{ ... }
```

- This has the same optional attributes as **@Stateless**.
- The stateful bean's lifecycle has a bit more to it, and involves two additional (optional) lifecycle hook methods:

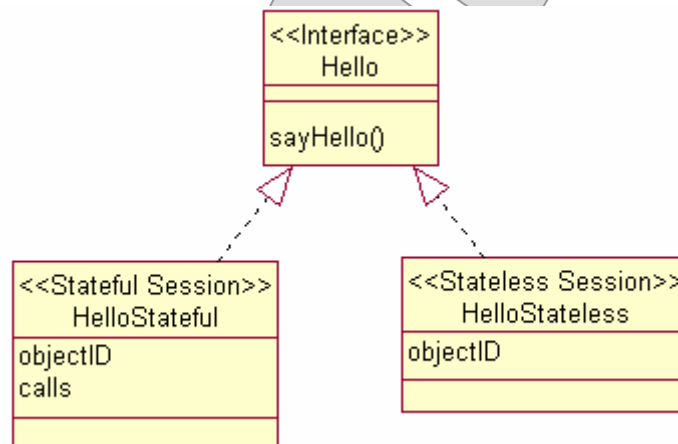


- **@PrePassivate** and **@PostActivate** allow the bean to perform passivation/activation tasks beyond the automatic save/reload that is carried out by the container.
- The state diagram above is simplified slightly; there are wrinkles having to do with the transactional status of a pooled bean.

Singletons and Pools

DEMO

- We'll work through a demonstration of singleton and pool behavior for stateful and stateless session beans, in **Demos/Lifecycle**.
 - We'll also add lifecycle hook methods and see that they're called.
 - The completed demo is in **Examples/Hello/Step2**.
- The starter code defines two session beans, both implementing the **Hello** interface:



Singletons and Pools

DEMO

- Both beans do a little trick that is not good practice for EJBs in general, but will be useful to us in spying on the container a little bit: they use **static** counters to assign serial IDs to objects as they are created.

– See `src/cc/ejb/HelloStateless.java`:

```
public HelloStateless ()
{
    objectID = ++instanceCount;
}

public String sayHello ()
{
    return "Hello! (Object " + objectID + ")";
}

private static int instanceCount = 0;
private int objectID;
```

Singletons and Pools

DEMO

- Then, the stateful bean also keeps a count of calls to **sayHello**, and responds differently as the method is called repeatedly by a specific client.

– See `src/cc/ejb/HelloStateful.java`:

```
public String sayHello ()
{
    String result = null;
    if (++calls == 1)
        result = "Hello!";
    else
    {
        result = "I already said, \"Hello\"";
        if (calls > 2)
            result += " " + (calls - 1) + " times!";
        else
            result += " once.";
    }

    return String.format ("% -34s", result) +
        "(Object " + objectID + ")";
}

private int calls;
```

- The client application in `src/Client.java` creates two instances of each bean type and then calls each one four times.

Singletons and Pools

DEMO

1. Build and deploy the EJB application – this command also builds a separate application client JAR around the **Client** class:

asant

2. Test by launching the application client:

asant run-appclient

Stateful reference #1:

```
Hello! (Object 1)
I already said, "Hello" once. (Object 1)
I already said, "Hello" 2 times! (Object 1)
I already said, "Hello" 3 times! (Object 1)
```

Stateful reference #2:

```
Hello! (Object 2)
I already said, "Hello" once. (Object 2)
I already said, "Hello" 2 times! (Object 2)
I already said, "Hello" 3 times! (Object 2)
```

Stateless reference #1:

```
Hello! (Object 1)
Hello! (Object 1)
Hello! (Object 1)
Hello! (Object 1)
```

Stateless reference #2:

```
Hello! (Object 1)
Hello! (Object 1)
Hello! (Object 1)
Hello! (Object 1)
```

Singletons and Pools

DEMO

- What does this output tell us?
 - Notice that there are **different object IDs** for the two **stateful beans**; this indicates that each new reference by **Client** causes a distinct session bean to be instantiated.
 - Conversely, only **one stateless bean** is created, even though there are two references: this is a singleton, not a growing pool.
 - Run the application a second time to see more of the same behavior, and notice that the container creates objects 3 and 4 for this “new” pair of client references.
 - And, we see that the stateful bean does indeed hold client state, as each bean instance holds a distinct count of calls from its client.
- 3. Add a lifecycle hook method to the stateful bean, to override the way in which the object ID is assigned:

```
@PostConstruct
public void postConstruct ()
{
    objectID = (int)
        (System.currentTimeMillis () & 0x7FFFFFFF);
    try { Thread.sleep (250); }
        catch (InterruptedException ex) {};
}
```

- 4. Add the same method to the stateless source file.

Singletons and Pools

DEMO

5. Build and test again, and see that the hook methods are correctly invoked by the container:

asant run-appclient

Stateful reference #1:

```
Hello! (Object 1883913306)
I already said, "Hello" once. (Object 1883913306)
I already said, "Hello" 2 times! (Object 1883913306)
I already said, "Hello" 3 times! (Object 1883913306)
```

Stateful reference #2:

```
Hello! (Object 1883913556)
I already said, "Hello" once. (Object 1883913556)
I already said, "Hello" 2 times! (Object 1883913556)
I already said, "Hello" 3 times! (Object 1883913556)
```

Stateless reference #1:

```
Hello! (Object 1883913837)
Hello! (Object 1883913837)
Hello! (Object 1883913837)
Hello! (Object 1883913837)
```

Stateless reference #2:

```
Hello! (Object 1883913837)
Hello! (Object 1883913837)
Hello! (Object 1883913837)
Hello! (Object 1883913837)
```

Initializing Stateful Beans

- EJB 2.1 specified home interfaces for beans, and these could define one or more **creation methods**, much the way one would write one or more overloaded constructors for a class.
- EJB 3.0 does away with the home interface, except for compatibility with EJB 2.1.
- It is possible to annotate a creation method with **@Init**.
 - This too is meant for backward compatibility, and is not recommended as best practice for EJB 3.0.
 - To use this annotation one must also define a **@Home** interface for the bean itself, and there's a domino effect that brings additional development tasks at that point.
- For 3.0 session beans, any conversational state should be provided through business methods.

- The following method signatures suggest a stateful use case:

```
public List<Car> showAvailableRentalCars  
    (Calendar startDate, Calendar endDate);  
public void selectCar (Car car);  
public void reserve (String myName, String email);
```

- Often a bean will define a business method such as **init**, which will be called by one party to set up a stateful bean for use by another party.

```
public void init (Department department);
```

The ProposalService

LAB 3B**Optional**

Suggested time: 30-60 minutes

In this lab you will implement the **ProposalService** stateful bean for the LandUse application. This bean wraps a single **Proposal** object for any number of operations by the caller. You will also refactor **LandUseServiceImpl** so that, instead of returning **Proposal** objects directly, it hands out newly-created instances of the stateful bean, and lets the client deal with this bean.

Detailed instructions are found at the end of the chapter.

Evaluation Only

SUMMARY

- **Session beans are probably the most natural and intuitive of the bean types.**
 - They are as close as we can get to a simple Java interface and class, while enjoying the features of an enterprise container such as remote connectivity, scalability, and security.
 - Building a session bean can be as simple as applying the **@Stateless** annotation to the implementation class.
- **It's important to choose carefully between stateful and stateless modes:**
 - The client code can be simpler when working with a stateful bean over several related requests.
 - There is a significant performance cost due to the need to pool stateful objects and to save off their conversational state.

The LandUseService

LAB 3A

In this lab you will implement the **LandUseService** stateless session bean. This will enable part of the web interface – by the end of the lab the user will be able to view a summary of all proposals, navigate to a non-editable detail view of any one proposal, and add and remove proposals. Other functions will come online in a later lab when you build the **ProposalService**.

Lab workspace:	Labs/Lab3A
Backup of starter code:	Examples/LandUse/Step1
Answer folder(s):	Examples/LandUse/Step2
Files:	src/gov/usda/usfs/landuse/ejb /LandUseService.java (to be created) src/gov/usda/usfs/landuse/ejb /LandUseServiceImpl.java (to be created) src/gov/usda/usfs/landuse/web/ProposalsServlet.java src/gov/usda/usfs/landuse/web/ProposalServlet.java META-INF/application.xml

Instructions:

1. Review the domain classes in the directory **src/gov/usda/usfs/landuse**, and compare to the UML model you've just seen.
2. Create the source file for a new interface **gov.usda.usfs.landuse.ejb.LandUseService**.
3. Define methods as shown below on the interface – this allows the caller to get all proposals; get a single proposal by ID; add and remove proposals; and there is a special method allowing a caller to set the full model of Java objects into the session bean. (This last method is temporary; it will go away when we start using entities.)

```
public List<Proposal> getAll ();
public Proposal getProposal (int ID);
public void submit (Proposal proposal);
public void withdraw (int ID);
public void primeWithData (List<Proposal> DB);
```

4. Add **import** statements for the **Proposal** class and **java.util.List**.
5. Now create a source file for the implementation class **LandUseServiceImpl**. Annotate it as a **@Stateless** bean. Make it implement the interface and create empty method definitions for all the interface methods – we'll work through the method implementations over the next few steps.
6. Declare a private field **DB** whose type is **List<Proposal>**.

The LandUseService**LAB 3A**

7. Implement **getAll** to return **DB**.
8. Implement **primeWithData** to set **DB** to the given list of proposals. Then loop through the list and set the ID of each proposal to its position in the list, plus one.
9. Run **asant build** now to compile your class and check your syntax.
10. Open **ProposalsServlet.java**. At the bottom of the file, declare a dependency on the **LandUseService** using the **@EJB** annotation – call it **landUseService**. (Remember to import this annotation from the **javax.ejb** package, and to import **LandUseService** itself.)
11. In the **init** method, there is already code to load the “database” from a file. At the end of the **try** block, just call **primeWithData** on your service reference.
12. In **doGet**, at the end of the long **try** block, but just before forwarding to **summary.jsp**, add a call to **getAll** on the bean, and store the list of proposals as a request attribute named “proposals”.
13. Open **application.xml** and add a declaration of the new EJB module:

```
<module>
  <ejb>LandUse.jar</ejb>
</module>
```

14. Run **asant** to build and deploy the application. Test by directing your browser to the following URL: you should see the list of proposals in **summary.jsp**:

<http://localhost:8080/LandUse>

USFS Land-Use Proposals

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

Select	Applicant	Parcel	Proposal	Decision
<input type="checkbox"/>	Mines-R-Us	Tonto NF	Silver mining	Accepted
<input type="checkbox"/>	Cranmore Paper	White Mountains NF	Selective logging	Accepted
<input type="checkbox"/>	Ski USA	Green Mountain NF	Alpine park	

Add
Edit
Remove

Not much else is working yet, though. Now we'll start to enable additional features by implementing service methods and calling them from the servlets.

The LandUseService**LAB 3A****Optional Steps**

15. Implement **LandUseServiceImpl.getProposal** to loop through the proposal objects in **DB**, find the one with an ID that matches the method parameter, and return that. Throw an **IllegalArgumentException** if the loop falls through without finding a proposal.
16. In **ProposalsServlet.doGet**, find the block of code that handles the “edit” command, and see that it already works through the request parameters to determine which proposal the user wants to edit. At the **TODO-3A** comment, call **getProposal** on the bean, passing the local variable **ID**, and store the results in a session attribute called “proposal”.
17. Build and test, and you should see that you can navigate from the summary to a detail page for a specific proposal by checking a box in the summary and clicking **Edit**.

Proposal	Public Comments								
Affected parcel: <input type="text" value="White Mountains NF"/> Applicant: <input type="text" value="Cranmore Paper"/> Application date: <input type="text" value="10/15/06"/> Proposed use: <input type="text" value="Selective logging"/> Proposed start date: <input type="text" value="7/15/07"/> Proposed end date: <input type="text" value="11/30/07"/> <input type="button" value="Done"/>	<table border="1"> <thead> <tr> <th>Contributor</th> <th>Recommendation</th> </tr> </thead> <tbody> <tr> <td>Zeldon Shelbow</td> <td>REJECT</td> </tr> </tbody> </table> <input type="button" value="Add a Public Comment"/>	Contributor	Recommendation	Zeldon Shelbow	REJECT				
Contributor	Recommendation								
Zeldon Shelbow	REJECT								
	<table border="1"> <thead> <tr> <th>Contributor</th> <th>Recommendation</th> </tr> </thead> <tbody> <tr> <td>Lanelle Thompson</td> <td>ACCEPT</td> </tr> </tbody> </table> <input type="button" value="Add a Professional Comment"/>	Contributor	Recommendation	Lanelle Thompson	ACCEPT				
Contributor	Recommendation								
Lanelle Thompson	ACCEPT								
<table border="1"> <thead> <tr> <th colspan="2">Decision</th> </tr> </thead> <tbody> <tr> <td>Decision:</td> <td>Accepted</td> </tr> <tr> <td>Date:</td> <td>11/30/07</td> </tr> <tr> <td>Authority:</td> <td>Jan Landry</td> </tr> </tbody> </table>	Decision		Decision:	Accepted	Date:	11/30/07	Authority:	Jan Landry	<input type="button" value="Cancel"/>
Decision									
Decision:	Accepted								
Date:	11/30/07								
Authority:	Jan Landry								

The LandUseService**LAB 3A**

18. Implement **LandUseServiceImpl.submit**: first, check that the submitted proposal has an ID of zero, and if it doesn't, throw an **IllegalArgumentException**. Then set the proposal ID to the current size of the **DB** list, plus one, and add the proposal to **DB**.
19. In **ProposalServlet.doGet** (the class name uses the singular **Proposal**, not the plural **Proposals**), un-comment the line of code that retrieves the session-scope "proposal".
20. Now uncomment the code that handles the "add" command – see that it creates a new **Proposal** object, calls the helper method **readInto** to populate it with request-parameter values, and calls **submit** on the bean before storing off the new object and forwarding to **summary.jsp**.
21. Build and test and see that you can add a new proposal by clicking **Add** on the summary page, filling in fields on the next page, and clicking **Submit**.
22. Implement **LandUseServiceImpl.withdraw** to call **getProposal** to find the proposal object for the given ID, and then remove it from **DB**.
23. Once again in **ProposalsServlet.doGet**, find the code that handles the "remove" command, call **withdraw** on the bean, passing the calculated **ID**.
24. Build and test one last time and confirm that you can remove proposals from the list by selecting one and clicking **Remove**.

The ProposalService

LAB 3B
Optional

In this lab you will implement the **ProposalService** stateful bean for the LandUse application. This bean wraps a single **Proposal** object for any number of operations by the caller. You will also refactor **LandUseServiceImpl** so that, instead of returning **Proposal** objects directly, it hands out newly-created instances of the stateful bean, and lets the client deal with this bean.

Lab workspace: Labs/Lab3B

Backup of starter code: Examples/LandUse/Step2

Answer folder(s): Examples/LandUse/Step3

Files:

- `src/gov/usda/usfs/landuse/ejb`
`/ProposalService.java` (to be created)
- `src/gov/usda/usfs/landuse/ejb`
`/ProposalServiceImpl.java` (to be created)
- `src/gov/usda/usfs/landuse/ejb/LandUseService.java`
- `src/gov/usda/usfs/landuse/ejb`
`/LandUseServiceimpl.java`
- `src/gov/usda/usfs/landuse/web/ProposalsServlet.java`
- `src/gov/usda/usfs/landuse/web/ProposalServlet.java`
- `src/gov/usda/usfs/landuse/web`
`/AddCommentServlet.java`

Instructions:

1. Define the interface **ProposalService** with the following method signatures:

```
public void init (Proposal model);
public Proposal get ();
public void update (Proposal proposal);
public void decide (Decision decision);
public void addPublicComment (Comment newComment);
public void addProfessionalComment (Comment newComment);
```

2. Create the implementation class **ProposalServiceImpl**. Annotate this as **@Stateful** and make it implement **ProposalService**. Create skeletons for all of the method signatures.
3. Run **asant build** to check your coding to this point.

The ProposalService

LAB 3B

This lab will follow a similar process to the previous lab: you'll implement a method or two, enable the servlet layer to use that method, and test a new feature of the application. First, though, we need to put the stateful bean in play, and that will mean refactoring the **LandUseServiceImpl** class: now, instead of having that service provide **Proposal** objects, we want it to provide new stateful-bean instances.

4. Open **LandUseService.java** and change the signature of the **getProposal** method: make it return a **ProposalService** reference, and change the method name to **getServiceForProposal**.
5. Open **LandUseServiceImpl.java** and declare an **@EJB** reference to a **ProposalService**.
6. Leave the **getProposal** method where it is, since it's also a helper method for other business methods. Create a new **getServiceForProposal** method to match the signature in the interface. In the method body, call **init** on the proposal service, passing the proposal object found by calling **getProposal**. Then return the **proposalService**.
7. Now we can get started implementing the stateful bean. Start by defining a private field called **model**, of type **Proposal**.
8. Implement **init**, which is the means by which the stateless bean can set the stateful bean's underlying proposal object. Simply set **model** to the given proposal object.
9. Implement **get** to return **model**.
10. In the **ProposalsServlet**, change the **getProposal** call to **getServiceForProposal**, and set the service and the proposal it represents as session attributes: "proposalService" and "proposal".
11. You can build and test at this point. There are no new behaviors in the application but you want to be sure that the refactoring you just implemented hasn't caused any regressions. You should still be able to view the summary and detail pages and to add and remove proposals.
12. Implement **ProposalServiceImpl.update** to transfer all six properties **affectedParcel**, **applicant**, **applicationDate**, **proposedUse**, **useStart**, and **useEnd** from the provided proposal object to the **model** object.
13. In **ProposalServlet**, un-comment the code that retrieves the stateful bean from the session object – near the top of the **doGet** method.
14. Now un-comment the code that handles the "done" command – see that it calls your **update** method, after reading request parameter values into a transfer object with the help of the **readInto** method.
15. Near the bottom of the **doGet** method, un-comment the line of code that removes the "proposalService" session attribute along with the others.

The ProposalService

LAB 3B

16. Build and test, and you should see that you can now edit the primary values on any proposal by navigating to the detail page for that proposal, making changes in the form fields, and clicking **Done**.

Optional Steps

17. Implement **ProposalServiceImpl.decide**. Start by checking that **model.getDecision** returns **null**. We don't allow a decision to be altered once it's made, so if this value is not **null**, throw an **IllegalStateException**. Then call **model.setDecision**, passing the given **Decision** object.
18. Un-comment the code in **ProposalServlet** that handles the "decide" command; this code reads request values into a transfer object and passes it to **decide**.
19. Build and test, and see that if you choose the third proposal in the summary – which has not yet been decided – and click **Edit**, you can enter a decision in the detail page, click **Decide**, and see it reflected in the summary page.
20. Finally, implement **addPublicComment** and **addProfessionalComment** on **ProposalServiceImpl**. For each method, call **getPublicComment** (or **getProfessionalComment**) and call **add** on the list of comments you get back, passing the given **Comment** object.
21. Open **AddCommentServlet.java**, and un-comment the code to derive the **ProposalService** reference from a session attribute, and also the code that calls one of the two **addXXXComment** methods on the bean.
22. Build and test and see that you can click **Add Public Comment** on a detail page, fill out the comment form, and click **Submit**, and see the comment appear on the detail page. Try the same test for professional comments.

A last thought: do you see anything wrong with our design regarding the relationship between these two session beans? To wit: how can multiple callers to the stateless session be assured that they're getting their own distinct instance of the stateful bean, that wraps the proposal in which they're interested? There's only one reference to the stateful bean, and it's getting re-initialized and handed out for each new call to **getServiceForProposal**. Can this work?

The answer is actually no – it can't. To correct this and make **LandUseServiceImpl** a factory for **ProposalServiceImpl** objects, we're going to need to learn more about dependency injection and move beyond these very simple **@EJB** references we've been using. For this lab, and for the next chapter or so, we're going to live with the limitation that only one user can work with the application at a time; then we'll correct this in a lab exercise in the dependency-injection chapter.