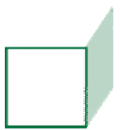




CHAPTER 3
SESSION BEANS



OBJECTIVES

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

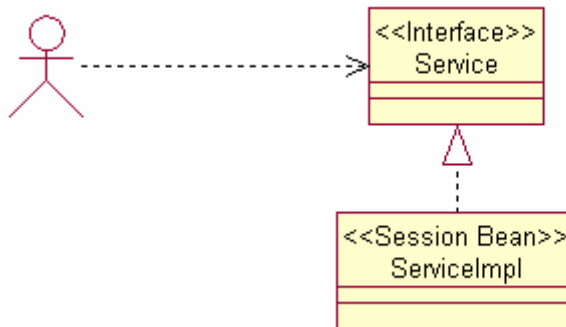
- Explain the functioning of stateful and stateless session beans.
- Describe the lifecycle and context interactions between container and session bean.
- Implement stateless session beans as interfaces to enterprise business logic.
- Understand the impact of the stateful/stateless choice on client usability 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 ...
```

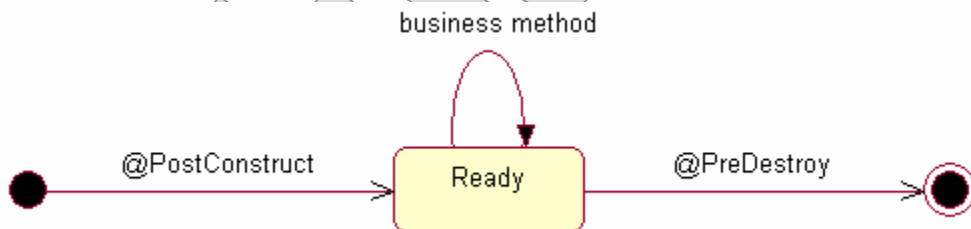
- **mappedName** defines a global JNDI name for the bean. This is only meaningful for a remote bean and it is non-portable – different application servers will handle JNDI differently.

```
@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

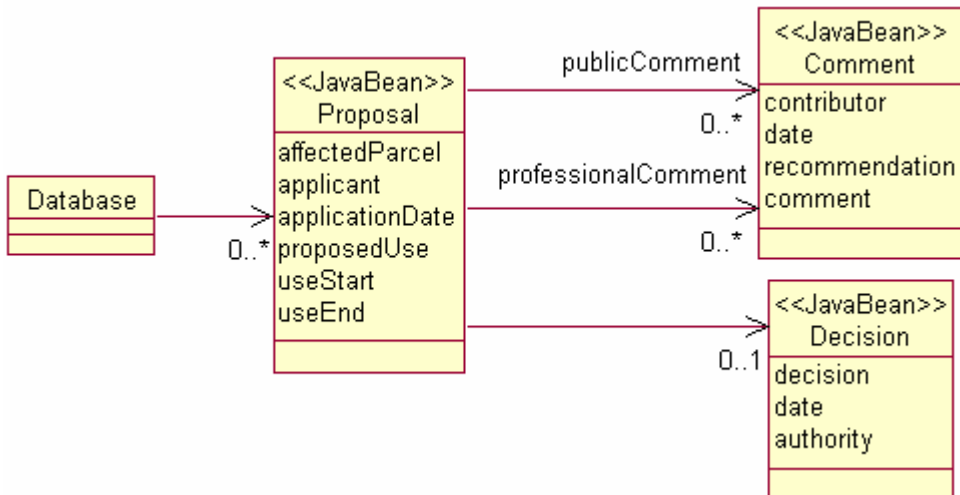
- In the previous chapter we spoke of lifecycle and context interfaces as the two channels of communication between the bean and the container.
- Lifecycle hook methods implement one channel, allowing the container to call the bean.
- The bean can call the container through the **SessionContext** interface.
- 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

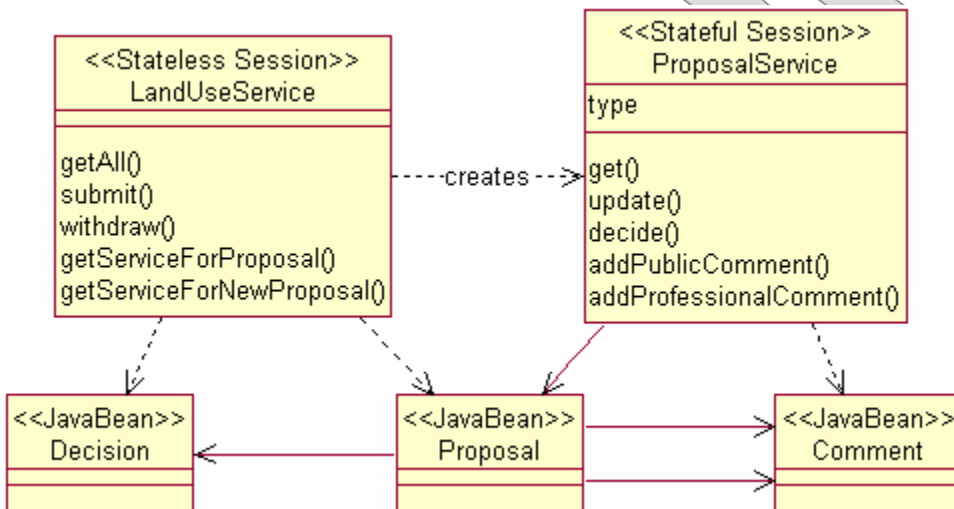
The LandUse Case Study

- 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.

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.

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

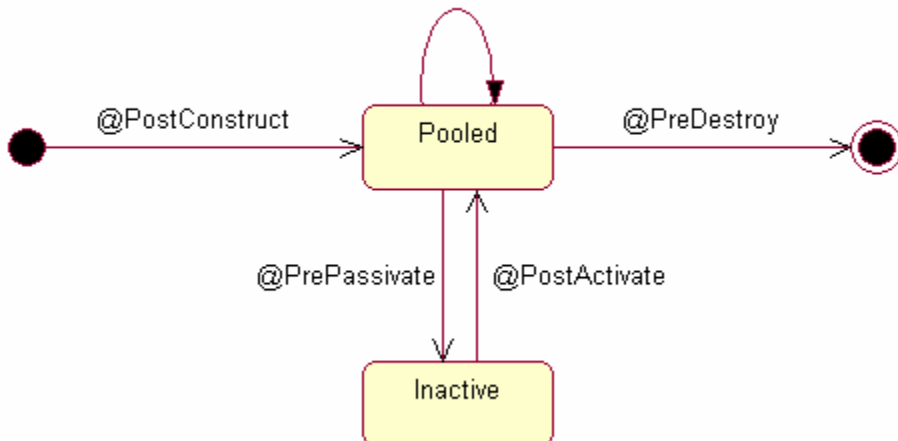
- Inform the container that it's dealing with a stateful bean using the **@Stateful** annotation:

```
@Stateful
```

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

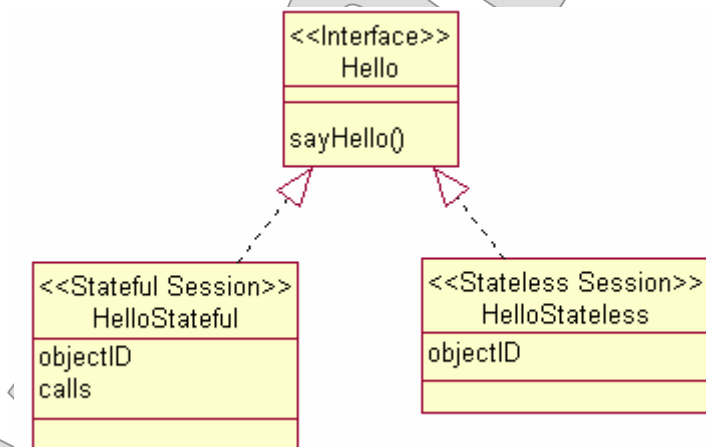
- This has the same optional attributes as **@Stateless**.
- The stateful bean's lifecycle has a bit more to it, and involves two additional (optional) lifecycle hook methods:

business 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.

- 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:



- 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;
```

- 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.

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

ant

2. Test by launching the application client, using a prepared script:

runAC

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)
```

- 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.

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

runAC

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);
```

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.

Evaluated
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.