Beginning EJB™ 3 Application Development

From Novice to Professional

Raghu R. Kodali and Jonathan Wetherbee
with Peter Zadrozny

Beginning EJB™ 3 Application Development: From Novice to Professional

Copyright © 2006 by Raghu R. Kodali and Jonathan Wetherbee with Peter Zadrozny

ISBN-13 (pbk): 978-1-59059-671-5

ISBN-10 (pbk): 1-59059-671-4

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com,` or visit `http://www.springeronline.com.`

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com,` or visit `http://www.apress.com.`

The source code for this book is available to readers at `http://www.apress.com` in the Source Code/ Download section. You will need to answer questions pertaining to this book in order to successfully download the code.

■ ■ ■

EJB 3 Session Beans

Introduction

This chapter will discuss EJB 3 session beans, the core business service objects used by EJB client applications. You'll gain an understanding of the new and simplified EJB 3 session bean model, with insight into the following topics:

- Types of session beans, both stateful and stateless, and when to use which

- The bean class, business interfaces, and business methods

- Callback methods

- Interceptors

- Exception handling

- Client view

- Dependency injection with annotations related to session beans

Introduction to Session Beans

Session beans are Java components that run in either stand-alone EJB containers or EJB containers that are part of standard Java Platform, Enterprise Edition (Java EE) application servers. These Java components are typically used to model a particular user task or use case, such as entering customer information or implementing a process that maintains a conversation state with a client application. Session beans can hold the business logic for many types of applications, such as human resources, order entry, and expense reporting applications.

Types of Session Beans

Session beans are of two types, as follows:

- *Stateless*: This type of bean does not maintain any conversational state on behalf of a client application.

- *Stateful*: This type of bean maintains state, and a particular instance of the bean is associated with a specific client request. Stateful beans can be seen as extensions to client programs that are running on the server.

We will drill down into more specifics of stateless and stateful beans in the following sections.

When Do You Use Session Beans?

Session beans are used to write business logic, maintain a conversation state for the client, and model back-end processes or user tasks that perform one or more business operations. Typical examples include the following:

- A session bean in a human resources application that creates a new employee and assigns the employee to a particular department

- A session bean in an expense reporting application that creates a new expense report

- A session bean in an order entry application that creates a new order for a particular customer

- A session bean that manages the contents of a shopping cart in an e-commerce application

- A session bean that leverages transaction services in an EJB 3 container (removing the need for an application developer to write the transaction support)

- A session bean used to address deployment requirements when the client applications are not colocated on the same server

- A session bean that leverages the security support provided by the container on the component or method level

Session beans can be used in traditional 2-tier or 3-tier architectures with professional/rich client applications, or in 3-tier web-based applications. These applications can be deployed in different logical and physical tier combinations. In the next section, we will investigate some of the possible combinations.

3-Tier Architecture with Rich Client

Figure 2-1 shows a typical architecture for a session bean in 3 tiers, with a rich client front-end application that has some data entry screens used by end users like customer service representatives, bank tellers, and so on. These client applications can be developed using Java Swing technology with the Java Platform, Standard Edition (Java SE), or they can be plain old Java objects (POJOs) that are run from the command line. Generally, the end user launches the client application from his desktop, enters some data, and triggers an event by pressing some user interface component such as a Submit button. The general workflow may look something like this:

1. User action establishes a connection to the session bean running in the EJB container using remote method invocation (RMI).

2. The client application invokes one or more business methods in the session bean.

3. The session bean processes the request and validates data—by interacting with databases, enterprise applications, legacy systems, and so on—to perform a certain business operation or task.

4. The session bean finally sends a response back to the client application, either through data collections or simple objects that contain acknowledgment messages.
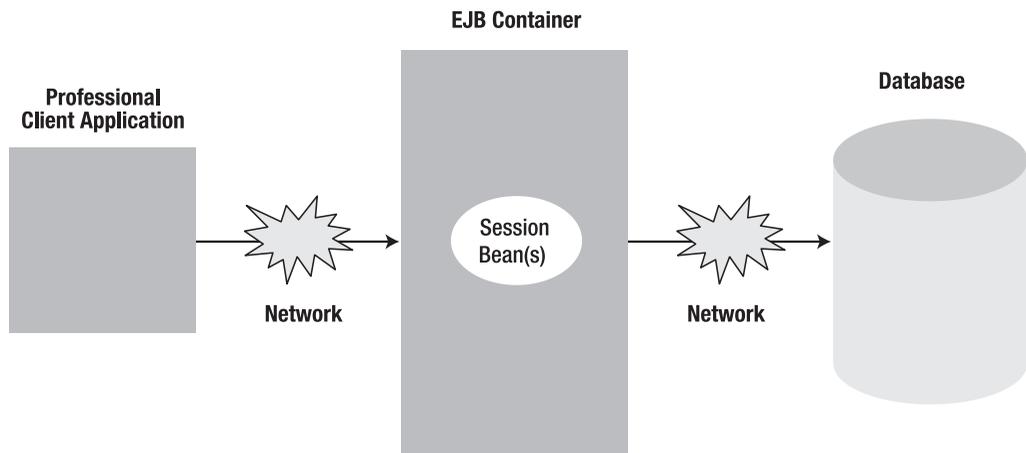
Figure 2-1. *Session beans in a 3-tier architecture*

3-Tier Architecture for a Web Application

This architecture, as shown in Figure 2-2, is typically front-ended by a web application running in the browser of a desktop or laptop machine. These days, other types of client devices, such as PDAs, cell phones, and telnet devices, are also being used to run these applications. The web application running in a browser or mobile device renders the user interface (data entry screens, Submit buttons, etc.) using web technologies such as JavaServer Pages (JSP), JavaServer Faces (JSF), or Java Servlets. Typical user actions, such as entering search criteria or adding certain items to the web application shopping cart, will invoke/call session beans running in an EJB container via one of the aforementioned web technologies. Once the session bean gets invoked, it processes the request and sends a response back to the web application, which formats the response as required, and then sends the response on to the requesting client device (browser, PDA, telnet, etc.).
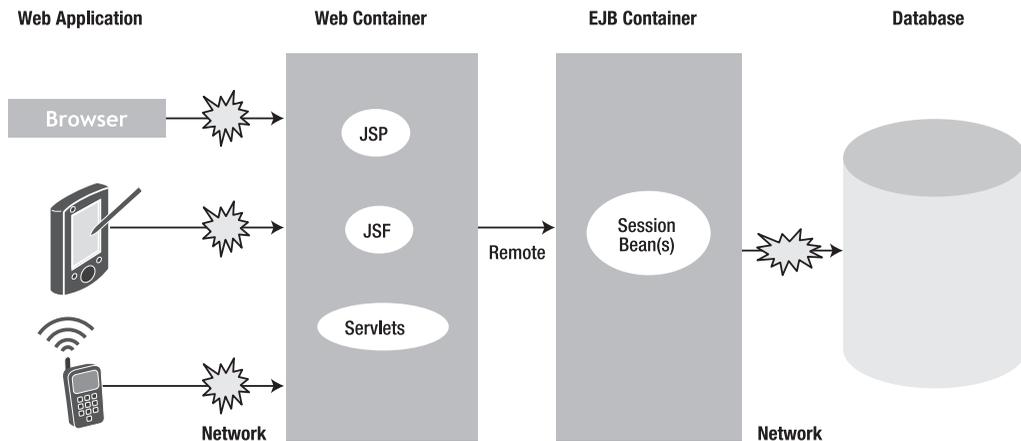
Figure 2-2. *Session beans in a 3-tier architecture with a web application*

In the 3-tier architecture just discussed, the client application (which is the web application) and the session beans can be run within the same instance of an application server (colocated) or from different instances running on the same machine. They can also be run in physically separate machines that have an instance of an application server.

Stateless Session Beans

Stateless session beans are comprised of the following elements:

- Business interfaces, which contain the declaration of business methods that are going to be visible to client applications

- A bean class, which contains the business method implementation to be executed

The Bean Class

A stateless session bean class is any standard Java class that has a class-level annotation of @Stateless. If deployment descriptors are used instead of annotations, then the bean class should be denoted as a stateless session bean. If you use both annotations and deployment descriptors (mixed mode), then the @Stateless annotation must be specified if any other class-level or member-level annotations are specified in the bean class. If both annotations and deployment descriptors are used, then the settings or values in the deployment descriptor will override the annotations in the classes during the deployment process.

To illustrate the use of stateless session beans, we will create a SearchFacade session bean that provides various search facilities to client applications regarding available wines. The workflow is as follows:

1. Users of the application will type in or choose one or more search criteria, which will be submitted to the SearchFacade session bean.

2. The SearchFacade bean will access back-end databases to retrieve the requested information. To simplify the code examples in this chapter, we will actually retrieve the list of hard-coded values within the bean class. In later chapters, we will augment the SearchFacade bean to access the back-end database.

3. The bean returns to the client applications the information that satisfied the search criteria.

Listing 2-1 shows the definition of the SearchFacade bean. In the following sections of this chapter, we will build the code that will show the preceding workflow in action. SearchFacadeBean is a standard Java class with a class-level annotation of @Stateless.

Listing 2-1. *SearchFacadeBean.java*

```
package com.apress.ejb3.chapter02;

import javax.ejb.Stateless;

@Stateless(name="SearchFacade")
public class SearchFacadeBean implements SearchFacade, SearchFacadeLocal {
    public SearchFacadeBean() {
    }
}
```

The Business Interface

A stateless session business interface is a standard Java interface that does not extend any EJB-specific interfaces. This interface has a list of business method definitions that will be available for the client application. Every session bean must have a business interface that can be implemented by the bean class, generated at design time by tools such as Oracle JDeveloper, NetBeans, or Eclipse; or generated at deployment time by the EJB container.

Business interfaces can use annotations as well, as described in the following list:

- The @Remote annotation can be used to denote the remote business interface.

- The @Local annotation can be used to denote the local business interface.

If no annotation is specified in the interface, then it is defaulted to the local interface.

If your architecture has a requirement whereby the client application (web application or rich client) has to run on a different Java Virtual Machine (JVM) from the one that is used to run the session beans in an EJB container, then you need to use the remote interface. The separate JVMs can be on the same physical machine or on separate machines. If your application architecture is going to use the same JVM for both the client application and the session beans, then use the local interface.

It is possible that your application architecture requires both remote and local interfaces. For example, an enterprise might have an order entry application that is developed using session beans that have business methods for submitting new orders and also addressing administrative tasks, such as data entry for the products. Potentially, you could have two different client applications that access the back-end order entry application, as follows:

- A web client application (as shown in Figure 2-3) that can be run in the same JVM as the session bean and used to submit new orders

- A rich client application (as shown in Figure 2-4) that runs on an end-user desktop machine and is used by the administrator for data entry purposes
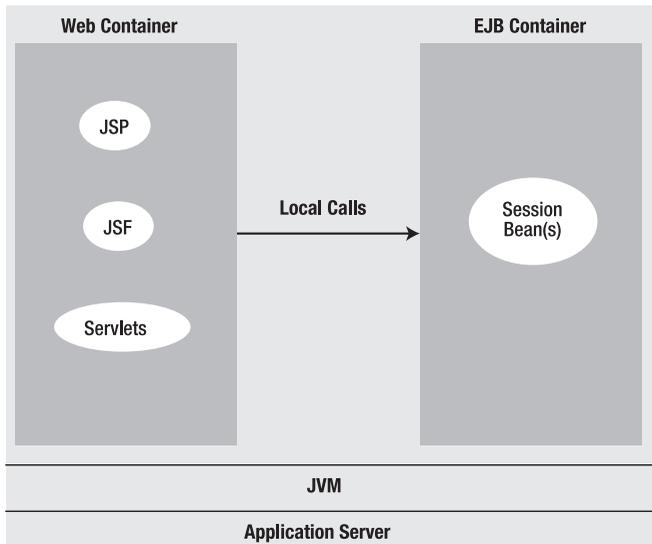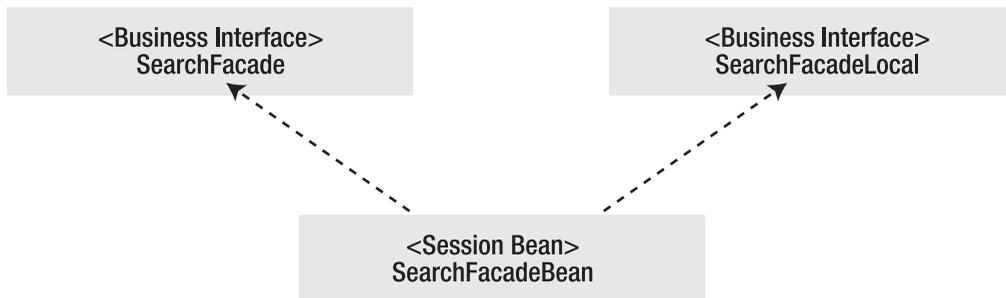

Figure 2-3. *A web client using local interfaces of session beans*
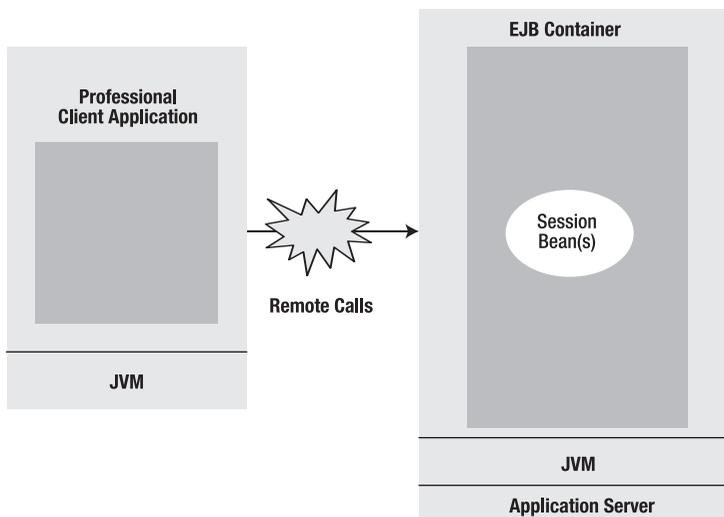
Figure 2-4. *A rich client using remote interfaces of session beans*

The `SearchFacade` session bean has both remote and local interfaces, as shown in Figure 2-5.

Figure 2-5. *The business interfaces of the SearchFacade session bean*

Listing 2-2 shows the code snippet for the `SearchFacade` remote business interface, with an `@Remote` annotation and a `wineSearch()` method declaration. The `wineSearch()` method takes one parameter that represents the type of the wine, and returns a list of wines that match the wine type criteria.

Listing 2-2. *SearchFacade.java*

```
package com.apress.ejb3.chapter02;

import java.util.List;

import javax.ejb.Remote;

@Remote
public interface SearchFacade {
    List wineSearch(String wineType);
}
```

Listing 2-3 shows the code snippet for the `SearchFacade` local business interface, with an `@Local` annotation and a `wineSearch()` method declaration.

Listing 2-3. *SearchFacadeLocal.java*

```
package com.apress.ejb3.chapter02;

import java.util.List;
```

```
import javax.ejb.Local;

@Local
public interface SearchFacadeLocal {
    List wineSearch(String wineType);
}
```

Business Methods

The methods implemented in the bean class must correspond to the business methods declared in the remote or local business interfaces. They are matched up based on the convention that they have the same name and method signature. Other methods in the bean class that do not have the corresponding declaration in the business interfaces will be private to the bean class methods.

The SearchFacade bean implements one method, wineSearch(), which has been declared in both remote and local business interfaces. The wineSearch() method returns a static wines list based on the type of wine. Listing 2-4 shows the implementation for wineSearch().

Listing 2-4. *SearchFacadeBean.java*

```
package com.apress.ejb3.chapter02;

import java.util.ArrayList;
import java.util.List;

import javax.ejb.Stateless;

@Stateless(name="SearchFacade")
public class SearchFacadeBean implements SearchFacade, SearchFacadeLocal {
    public SearchFacadeBean() {
    }

    public List wineSearch(String wineType) {
        List wineList = new ArrayList();
        if (wineType.equals("Red"))
            {
            wineList.add("Bordeaux");
            wineList.add("Merlot");
            wineList.add("Pinot Noir");
            }
```

```
        else if (wineType.equals("White"))
            {
            wineList.add("Chardonnay");

            }

        return wineList;
    }
}
```

Dependency Injection

In Chapter 1, we introduced the concept of dependency injection as a programming design pattern. In this section, we will look into using dependency injection in stateless session beans.

EJB 3 containers provide the facilities to inject various types of resources into stateless session beans. Typically, in order to perform user tasks or process requests from client applications, the business methods in the session bean require one or more types of resources. These resources can be other session beans, data sources, or message queues.

The resources that the stateless session bean is trying to use can be injected using annotations or deployment descriptors. Resources can be acquired by annotation of instance variables or annotation of the setter methods. Listing 2-5 shows an example of setter and instance variable–based injection of myDb, which represents the data source.

Listing 2-5. *Data Source Injection*

```
    @Resource
    DataSource myDb;
```

or

```
    @Resource
    public void setMyDb(DataSource myDb) {
        this.myDb = myDb;
    }
```

You typically use the setter injections to preconfigure or initialize properties of the injected resource.

Callback Methods

There will be certain instances or use cases in which the application using session beans requires fine-grained control over things like an object's creation, removal, and so on. For example, the SearchFacade session bean might need to perform some database initialization when it is created, or close some database connections when it is destroyed. The application can gain fine-grained control over the various stages of the bean life cycle via methods known as *callback methods*. A callback method can be any method in the session bean that has callback annotations. The EJB container calls these methods at the appropriate stages of the bean's life cycle (bean creation and destruction).

Following are two such callbacks for stateless session beans:

- PostConstruct: Denoted with the @PostContruct annotation. Any method in the bean class can be marked with this annotation.

- PreDestroy: Denoted with the @PreDestroy annotation. Again, any method in the bean class can be marked with this annotation.

PostContruct callbacks happen after a bean instance is instantiated in the EJB container. If the bean is using any dependency injection mechanisms for acquiring references to resources or other objects in its environment, PostConstruct will occur after injection is performed and before the first business method in the bean class is called.

In the case of the SearchFacade session bean, you could have a business method, wineSearchByCountry(), that would return the wine list for a particular country, and have a PostConstruct callback method, initializeCountryWineList(), that would initialize the country's wine list whenever the bean gets instantiated. Ideally, you would load the list from a back-end datastore; but in this chapter, we will just use some hard-coded values that get populated into a HashMap, as shown in Listing 2-6.

Listing 2-6. *The PostConstruct Method*

```
@PostConstruct
public void initializeCountryWineList()
    {
    //countryMap is HashMap
    countryMap.put("Australia", "Sauvignon Blanc");
    countryMap.put("Australia", "Grenache");
    countryMap.put("France","Gewurztraminer");
    countryMap.put("France","Bordeaux");
    }
```

The PreDestroy callback happens before the container destroys an unused or expired bean instance from its object pool. This callback can be used to close any connection pool that has been created with dependency injection, and also to release any other resources.

In the case of the SearchFacade session bean, we could add a PostConstruct callback method (destroyWineList()) into the SearchFacade bean, which would clear the country wine list whenever the bean gets destroyed. Ideally, during PostContruct, we would close any resources that have been created with dependency injection; but in this chapter, we will just clear the HashMap that has the countries and wine list. Listing 2-7 shows the destroyWineList() code.

Listing 2-7. *The PreDestroy Method*

```
@PreDestroy
public void destroyWineList()
    {
    countryMap.clear();
    }
```

Callback methods defined on a bean class should have the following signature:

```
public void <METHOD>()
```

Callback methods can also be defined on a bean's listener class; these methods should have the following signature:

```
public void <METHOD>(Object)
```

where Object may be declared as the actual bean type, which is the argument passed to the callback method at run time.

Interceptors

The EJB 3 specification provides annotations called *interceptors,* which allow you to intercept a business method invocation. An interceptor method can be defined for session and message-driven beans (MDBs). We will show you the usage of interceptors in the session bean context.

There are number of use cases for interceptors in a typical application, in which you would find a need to perform a certain task before or after the business method is invoked. For example, you may wish to do one of the following:

- Perform additional security checks before a critical business method that transfers more than $100,000 dollars

- Do some performance analysis to compute the time it takes to perform the task

- Do additional logging before or after the method invocation

You can either add an @AroundInvoke annotation on a particular method, or you can define an interceptor class whose methods are invoked before a business method is invoked in the bean class. An interceptor class is denoted by the @Interceptor annotation on the bean class with which it is associated. In the case of multiple interceptor classes, the @Interceptors annotation is used. Methods that are annotated with @AroundInvoke should have the following signature:

```
public Object <METHOD>(InvocationContext) throws Exception
```

The definition of InvocationContext is as follows:

```
package javax.ejb;

public interface InvocationContext {
    public Object getBean();
    public java.lang.reflect.Method getMethod();
    public Object[] getParameters();
    public void setParameters(Object[] params);
    public EJBContext getEJBContext();
    public java.util.Map getContextData();
    public Object proceed() throws Exception;
}
```

The following list describes the methods in the preceding code:

- getBean() returns the instance of the bean on which the method was called.

- getMethod() returns the method on the bean instance that was called.

- getParameters() returns the parameters for the method call.

- setParameters() modifies the parameters used for the method call.

- getEJBContext() gives the interceptor methods access to the bean's EJBContext.

- getContextData() allows values to be passed between interceptor methods in the same InvocationContext instance using the Map returned.

- proceed() invokes the next interceptor, if there is one, or invokes the target bean method.

In the `SearchFacade` session bean, we can add an interceptor that logs the time taken to execute each business method when invoked by the client applications. Listing 2-8 shows a time log method that will print out the time taken to execute a business method. `InvocationContext` is used to get the name of bean class and the invoked method name. Before invoking the business method, current system time is captured and deducted from the system time after the business method is executed. Finally, the details are printed out to the console log using `System.out.println`.

Listing 2-8. *The Interceptor Method*

```
@AroundInvoke
public Object TimerLog (InvocationContext ctx) throws Exception
    {
    String beanClassName = ctx.getClass().getName();
    String businessMethodName = ctx.getMethod().getName();
    String target = beanClassName + "." + businessMethodName ;
    long startTime = System.currentTimeMillis();
    System.out.println ("Invoking " + target);
        try {
            return ctx.proceed();
            }
        finally {
            System.out.println("Exiting " + target);
            long totalTime = System.currentTimeMillis() - startTime;
            System.out.println("Business method " + businessMethodName +
            "in " + beanClassName + "takes " + totalTime + "ms to execute");
            }
    }
```

Stateful Session Beans

Similar to stateless session beans, stateful beans comprise a bean class and a business interface.

The Bean Class

A stateful session bean class is any standard Java class that has a class-level annotation of `@Stateful`. If deployment descriptors are used instead of annotations, the bean class should be denoted as a stateful session bean. In the case of mixed mode, in which you are

using annotations *and* deployment descriptors, the `@Stateful` annotation must be speci-fied if any other class-level or member-level annotations are specified in the class.

To illustrate a stateful session bean, we will create a `ShoppingCart` session bean that will keep track of the items added to a user's shopping cart and their respective quanti-ties. In this chapter, we will use hard-coded values for the shopping cart to illustrate the state and conversation maintenance between the client and stateful session bean. Listing 2-9 shows the definition of a `ShoppingCart` session bean.

Listing 2-9. *ShoppingCartBean.java*

```
package com.apress.ejb3.chapter02;

import javax.ejb.Stateful;

@Stateful(name="ShoppingCart")
public class ShoppingCartBean implements ShoppingCart, ShoppingCartLocal {
    public ShoppingCartBean() {
    }
}
```

There will be certain use cases in which the application wants to be notified by the EJB container before or after transactions take place, and then use these notifications to manage data and cache. A stateful session bean can receive this kind of notification by the EJB container when it implements the `javax.ejb.SessionSynchronization` interface. This is an optional feature. There are three different types of transaction notifications that the stateful session bean receives from the EJB container, as follows:

- `afterBegin`: Indicates that a new transaction has begun

- `beforeCompletion`: Indicates that the transaction is going to be committed

- `afterCompletion`: Indicates that a transaction has been completed

For example, the `ShoppingCart` session bean could implement the `javax.ejb.SessionSynchronization` interface to get an `afterCompletion` notification, so that it can clear out the shopping cart cache.

The Business Interface

Business interfaces for stateful session beans are similar to those for stateless session beans, and are annotated in the same way, using `@Local` and `@Remote` annotations. The `ShoppingCart` session bean has both remote and local interfaces, as shown in Figure 2-6.
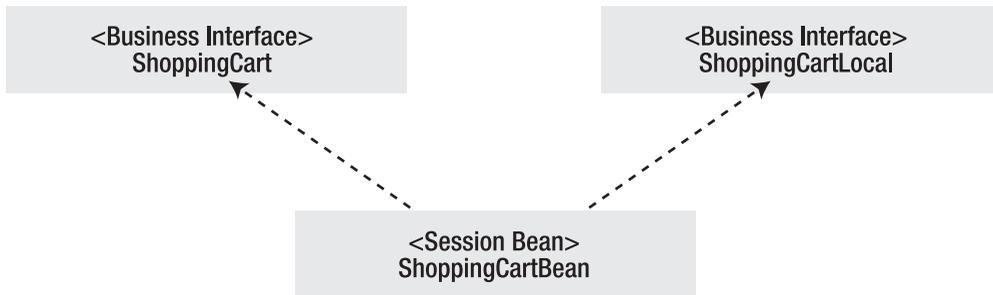
Figure 2-6. *Business interfaces for ShoppingCart*

We will primarily use the local interface from our web application. The remote interface is added to facilitate unit testing of the bean in this chapter.

Listings 2-10 and 2-11 show the remote and local ShoppingCart business interfaces, with @Remote and @Local annotations, respectively.

Listing 2-10. *ShoppingCart.java*

```
package com.apress.ejb3.chapter02;

import javax.ejb.Remote;

@Remote
public interface ShoppingCart {
}
```

Listing 2-11. *ShoppingCartLocal.java*

```
package com.apress.ejb3.chapter02;

import javax.ejb.Local;

@Local
public interface ShoppingCartLocal {
}
```

Alternatively, you can use the coding style shown in Listing 2-12, in which you can specify the @Local and @Remote annotations before specifying @Stateful or @Stateless with the name of the business interface.

Listing 2-12. *ShoppingCartBean.java*

```
package com.apress.ejb3.chapter02;



import javax.ejb.Local;
import javax.ejb.Remote;
import javax.ejb.Stateful;

@Local({ShoppingCartLocal.class})
@Remote({ShoppingCart.class})
@Stateful(name="ShoppingCart")

public class ShoppingCartBean implements ShoppingCart, ShoppingCartLocal {
    public ShoppingCartBean() {
    }
}
```

■**Note** In this book, we will follow the earlier convention, in which `@Local` and `@Remote` annotations are marked on the business interfaces.

Business Methods

Business methods in stateful session beans are similar to those in stateless session beans. We will augment the `ShoppingCart` bean by adding business methods that will add and remove wines from the shopping cart, and return a list of cart items.

Listing 2-13 shows the `ShoppingCart` bean implementing the `addWineItem()`, `removeWineItem()`, and `getCartItems()` methods.

Listing 2-13. *ShoppingCartBean.java*

```
package com.apress.ejb3.chapter02;
import java.util.ArrayList;

import javax.ejb.Stateful;

@Stateful(name="ShoppingCart")
```

```
public class ShoppingCartBean implements ShoppingCart, ShoppingCartLocal {
    public ShoppingCartBean() {
    }
    public ArrayList cartItems;

    public void addWineItem(String wine) {
       cartItems.add(wine);
    }

    public void removeWineItem(String wine) {
       cartItems.remove(wine);
    }

    public void setCartItems(ArrayList cartItems) {
        this.cartItems = cartItems;
    }

    public ArrayList getCartItems() {
        return cartItems;
    }
}
```

Callback Methods

Stateful session beans support callback events for construction, destruction, activation, and passivation. Following are the callbacks that map to the preceding events:

- PostConstruct: Denoted with the @PostConstruct annotation. Any method in the bean class can be marked with this annotation.

- PreDestroy: Denoted with the @PreDestroy annotation.

- PreActivate: Denoted with the @PreActivate annotation.

- PrePassivate: Denoted with the @PrePassivate annotation.

The PostContruct callback happens after a bean instance is instantiated in the EJB container. If the bean is using any dependency injection mechanism for acquiring references to resources or other objects in its environment, the PostConstruct event happens after injection is performed and before the first business method in the bean class is called.

In the case of the ShoppingCart session bean, we could have a business method called initialize() that initializes the cartItems list, as show in Listing 2-14.

Listing 2-14. *The PostConstruct Method*

```
@PostConstruct
public void initialize()
    {
    cartItems = new ArrayList();
    }
```

The PreDestroy callback happens after any method with an @Remove annotation has been completed. In the case of the ShoppingCart session bean, we could have a business method called exit() that writes the cartItems list into a database. In this chapter, we will just print out a message to the system console to illustrate the callback. Listing 2-15 shows the code for the exit() method, which has the @PreDestroy annotation.

Listing 2-15. *The PreDestroy Method*

```
@PreDestroy
public void exit()
    {
        // items list into the database.
        System.out.println("Saved items list into database");
    }
```

The @Remove annotation is a useful life cycle method for stateful session beans. When the method with the @Remove annotation is called, the container will remove the bean instance from the object pool after the method is executed. Listing 2-16 shows the code for the stopSession() method, which has the @Remove annotation.

Listing 2-16. *The Remove Method*

```
@Remove
public void stopSession()
    {
    // The method body can be empty.
    System.out.println("From stopSession method with @Remove annotation");
    }
```

The `PrePassivate` callback kicks in when a stateful session bean instance is idle for too long. During this event, the container might passivate and store its state to a cache. The method tagged with `@PrePassivate` is called before the container passivates the bean instance.

The `PostActivate` event gets raised when the client application uses a passivated stateful session bean again. A new instance with restored state is created. The method with the `@PostActivate` annotation is called when the bean instance is ready.

Interceptors

There are some minor differences between interceptors for stateless and stateful session beans. `AroundInvoke` methods can be used with stateful session beans. For stateful session beans that implement `SessionSynchronization`, `afterBegin` occurs before any methods that have `AroundInvoke` annotations, and before the `beforeCompletion()` callback method.

Exception Handling

The EJB 3 specification outlines two types of exceptions:

- Application exceptions

- System exceptions

Application exceptions are exceptions related to execution of business logic that the client should handle. For example, an application exception might be raised if the client application passes an invalid argument, such as the wrong credit card number.

System exceptions, on the other hand, are caused by system-level faults, such as Java Naming and Directory Interface (JNDI) errors, or failure to acquire a database connection. A system exception must be a subclass of a `java.rmi.RemoteException`, or a subclass of a `java.lang.RuntimeException` that is not an application exception.

From the EJB application point of view, application exceptions are done by writing application-specific exception classes that subclass the `java.lang.Exception` class.

In the case of a system exception, the application catches particular exceptions—such as a `NamingException` that results from a JNDI failure—and throws an `EJBException`. In this particular chapter, our examples aren't using any resources as such—but there are more examples of system exceptions in the later chapters.

Client View for Session Beans

A session bean can be seen as a logical extension of a client program or application, where much of the logic and data processing for that application happens. A client

application typically accesses the session object through the session bean's client view interfaces, which are the business interfaces that were discussed in the earlier sections.

A client application that accesses session beans can be one of three types:

Remote: Remote clients run in a separate JVM from the session beans that they access, as shown in Figure 2-4. A remote client accesses a session bean through the bean's remote business interface. A remote client can be another EJB, a Java client program, or a Java servlet. Remote clients have location independence, meaning that they can use the same API as the clients running in the same JVM.

Local: Local clients run in the same JVM, as shown in Figure 2-3, and access the session bean through the local business interface. A local client can be another EJB, or a web application using Java Servlets, JavaServer Pages (JSP), or JavaServer Faces (JSF). Local clients are location dependent. Remote and local clients are compared in Table 2-1.

Web Services: You can publish stateless session beans as web services that can be invoked by Web Services clients. We will discuss Web Services and clients in Chapter 6.

Table 2-1. *Considerations for Choosing Between Local and Remote Clients*

Remote	Local
Loose coupling between the bean and the client	Lightweight access to a component
Location independence	Location dependence
Expensive remote calls	Must be colocated with the bean
Objects must be serialized	Not required
Objects are passed by value	Objects are passed by reference

In some cases, the session beans need to have both local and remote business interfaces to support different types of client applications. A client can obtain a session bean's business interface via dependency injection or JNDI lookup. Before invoking the methods in the session bean, the client needs to obtain a stub object of the bean via JNDI. Once the client has a handle to the stub object, it can call the business methods in the session bean. In the case of a stateless session bean, a new stub can be obtained on every invocation. In the case of a stateful session bean, the stub needs to be cached on the client side so that the container knows which instance of the bean to return on subsequent calls. Using dependency injection, we can obtain the business interface of the SearchFacade session bean with the following code:

```
@EJB SearchFacade searchFacade;
```

If the client accessing the session bean is remote, the client can use JNDI lookup once the context interface has been obtained with the right environment properties. Local clients can use JNDI lookup as well, but dependency injection results in simpler code. Listing 2-17 shows the SearchFacadeTest client program's JNDI code, which looks up the SearchFacade bean, invokes the wineSearch() business method, and prints out the returned list of wines.

■**Note** If the remote client is a Java application or command-line program, an application client container can be used to invoke the session beans. Application client containers support dependency injection for remote clients. We will discuss application client containers in Chapter 12, along with other types of client applications.

Listing 2-17. *SearchFacadeClient.java*

```java
package com.apress.ejb3.chapter02.client;

import com.apress.ejb3.chapter02.SearchFacade;
import java.util.List;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class SearchFacadeTest {
    public SearchFacadeTest() {
    }

    public static void main(String[] args) {
        SearchFacadeTest searchFacadeTest = new SearchFacadeTest();
        searchFacadeTest.doTest();
    }

    @EJB
    static SearchFacade searchFacade;

    void doTest(){
        InitialContext ic;

        try {
            ic = new InitialContext();
            System.out.println("SearchFacade Lookup");
```

```
            System.out.println("Searching wines");
            List winesList = searchFacade.wineSearch("Red");
            System.out.println("Printing wines list");
            for (String wine:(List<String>)winesList ){
                System.out.println(wine);
            }
        } catch (NamingException e) {
            e.printStackTrace();
        }
    }
}
```

Listing 2-18 shows the ShoppingCartTest client program, which looks up the stateful ShoppingCart session bean, calls the addWineItem() business method to add a wine to the shopping cart, calls the getCartItems() business method to get the items in the cart, and finally prints the list of wines in the shopping cart.

Listing 2-18. *ShoppingCartTest.java*

```
package com.apress.ejb3.chapter02.client;

import com.apress.ejb3.chapter02.ShoppingCart;

import java.util.ArrayList;

import java.util.List;

import javax.naming.InitialContext;
import javax.naming.NamingException;

public class ShoppingCartTest {
    public ShoppingCartTest() {
    }

    public static void main(String[] args) {
        ShoppingCartTest shoppingCartTest = new ShoppingCartTest();
        shoppingCartTest.doTest();
    }

@EJB
static ShoppingCart shoppingCart;
```

```
    void doTest(){
        InitialContext ic;

        try {
            ic = new InitialContext();
            System.out.println("ShoppingCart Lookup");
            System.out.println("Adding Wine Item");
            shoppingCart.addWineItem("Zinfandel");
            System.out.println("Printing Cart Items");
            ArrayList cartItems = shoppingCart.getCartItems();
            for (String wine:(List<String>)cartItems ){
                System.out.println(wine);
            }
        } catch (NamingException e) {
            e.printStackTrace();
        }
    }
}
```

Packaging, Deploying, and Testing the Session Beans

Session beans need to be packaged into EJB JAR (`.jar`) files before they are deployed into EJB containers. In the case of some EJB containers or application servers, packaged EJB archives need to be assembled into Enterprise Archive (EAR) files before deployment. EJB containers or application servers provide deployment utilities or Ant tasks to facilitate deployment of EJBs. Java IDEs (integrated development environments) like Oracle JDeveloper, NetBeans, and Eclipse also provide deployment features that allow developers to package, assemble, and deploy EJBs to application servers. Packaging, assembly, and deployment are covered in detail in Chapter 10.

So far in this chapter, we have developed one stateless session bean (`SearchFacade`) and one stateful session bean (`ShoppingCart`). The following sections will walk you through the steps to package, assemble, deploy, and test these session beans.

Prerequisites

Before performing any of the steps detailed in the next sections, complete the "Getting Started" section of Chapter 1, which will walk you through the installation and environment setup required for the samples in this chapter.

■**Note** We assume that the source code for this chapter's samples is located in the Z: drive. Replace Z: with the location of the directory into which you have downloaded the source.

Compiling the Session Beans

From the DOS console, execute the following javac command to compile the SearchFacade and ShoppingCart session beans, along with their business interfaces. Figure 2-7 shows the command being executed from the z:\Chapter02-SessionSamples\SessionBeanSamples directory.

```
Z:\Chapter02-SessionSamples\SessionBeanSamples>%JAVA_HOME%/bin/javac –classpath ➥
 %GLASSFISH_HOME%\lib\javaee.jar -d ./classes src\com\apress\ejb3\chapter02\*.java
```
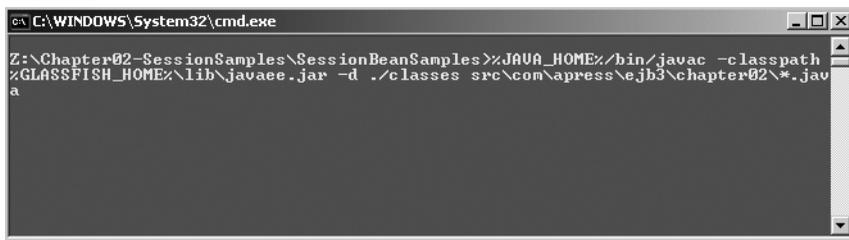

Figure 2-7. *Compiling the session beans*

In the downloaded source code for this chapter's samples, you will see the Ant build script (build.xml). You can alternatively use the following Ant task to compile the session beans:

```
Z:\Chapter02-SessionSamples\SessionBeanSamples>%ANT_HOME%/bin/ant ➥
Compile-SessionSamples
```

■**Note** If you are using the Ant task for compiling, make appropriate changes to the build.properties file to reflect the settings for the GlassFish application server that you have installed.
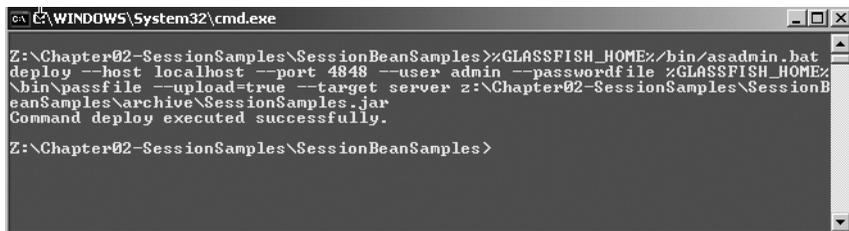
Packaging the Session Beans

Once the source code for the session beans is compiled, you need to package the compiled classes into an EJB JAR file. Execute the following command from the DOS console:

```
Z:\Chapter02-SessionSamples\SessionBeanSamples\classes>%JAVA_HOME%/bin/jar cvf ➥
 .. \archive\SessionSamples.jar .\com
```

■**Note** The preceding command is run from the classes directory.

Figure 2-8 shows the command being executed from the Z:\Chapter02-SessionSamples\ SessionBeanSamples\classes directory, and the files being added to the generated JAR file.

Figure 2-8. *Packaging the session beans*

Alternatively, you can use the following Ant task to package the session beans:

```
Z:\Chapter02-SessionSamples\SessionBeanSamples>%ANT_HOME%/bin/ant ➥
 Package-SessionBeans
```

The generated SessionSamples.jar file will be stored in the archive directory.

■**Note** If you are using the Ant task for packaging, make the appropriate changes to the build.properties file to reflect the settings for the GlassFish application server you have installed.

Deploying the Session Beans

Once you have compiled and packaged the session beans, you can deploy the generated JAR file (SessionSamples.jar) to the GlassFish application server. You can deploy the JAR file from the GlassFish administration console—however, for practical purposes, you'll want to automate these tasks as much as possible. You should use the following command-line utilities and Ant task for deployment.

From the DOS console, execute the following command:

```
Z:\Chapter02-SessionSamples\SessionBeanSamples>%GLASSFISH_HOME%/bin/asadmin.bat ➥
deploy --host localhost --port 4848 --user admin --passwordfile %GLASSFISH_HOME%\ ➥
asadminpass --upload=true --target server ➥
Z:\Chapter02-SessionSamples\SessionBeanSamples\archive\SessionSamples.jar
```

Figure 2-9 shows the command being executed from the z:\Chapter02-SessionSamples\SessionBeanSamples directory.

Figure 2-9. *Deploying the session beans*

Alternatively, you can use the following Ant task to deploy the session beans:

```
Z:\Chapter02-SessionSamples\SessionBeanSamples>%ANT_HOME%/bin/ant ➥
Deploy-SessionBeans
```

■**Note** If you are running the GlassFish application server on a different machine, replace localhost with that machine name in the command-line arguments. Similarly, if you are running on a different port, replace 4848 with the number of the port that you are using. If you are using the Ant task for deployment, make the appropriate changes to the build.properties file to reflect the settings for the GlassFish application server you have installed.
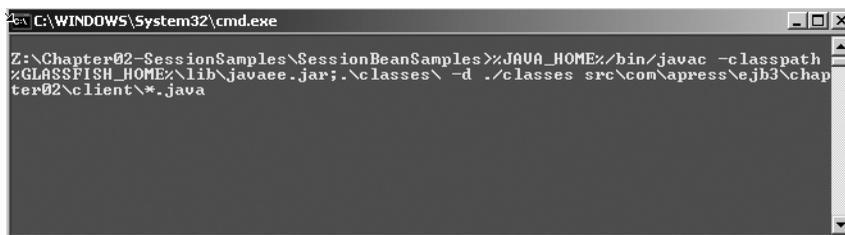

Compiling the Client Programs

From the DOS console, execute the following `javac` command to compile the `ShoppingCartTest` and `SearchFacadeTest` classes.

```
Z:\Chapter02-SessionSamples\SessionBeanSamples>%JAVA_HOME%/bin/javac –classpath ➥
%GLASSFISH_HOME%\lib\javaee.jar;.\classes\ -d ./classes ➥
 src\com\apress\ejb3\chapter02\client\*.java
```

Figure 2-10 shows the command being executed from the `Z:\Chapter02-SessionSamples\SessionBeanSamples` directory.

Figure 2-10. *Compiling the client programs*

Alternatively, you can use the following Ant task to compile the client programs:

```
Z:\Chapter02-SessionSamples\SessionBeanSamples>%ANT_HOME%/bin/ant ➥
Compile-ClientPrograms
```

Running the Client Programs

Once the client programs are compiled, you can invoke the business methods in the deployed `ShoppingCart` and `SearchFacade` session beans using the GlassFish application client container that supports dependency injection. You will assemble the client and its dependent classes into a JAR file, which will enable you to specify the JAR file as an argument to the application client container.
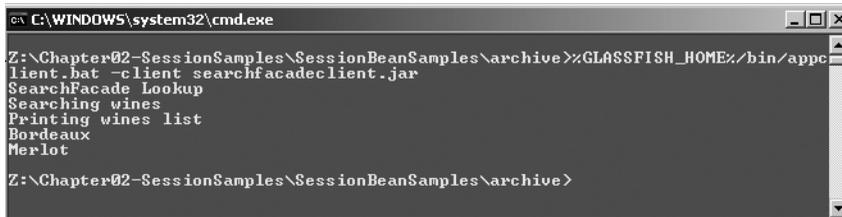
■**Note**   The application client container will be covered in detail in Chapter 12.

Execute the following command in the DOS shell to run SearchFacadeTest:

```
Z:\Chapter02-SessionSamples\SessionBeanSamples\archive>%GLASSFISH_HOME%/bin/➥
appclient.bat -client searchfacadeclient.jar
```

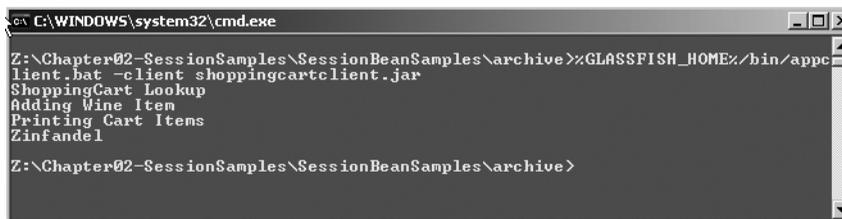Figure 2-11 shows the output printed to the console after successful execution of the client program.

Figure 2-11. *The SearchFacadeTest client results*

Execute the following command in the DOS shell to run ShoppingCartTest:

```
Z:\Chapter02-SessionSamples\SessionBeanSamples\archive>%GLASSFISH_HOME%/bin/➥
appclient.bat -client shoppingcartclient.jar
```

Figure 2-12 shows the output printed to the console after successful execution of the client program.

Figure 2-12. *The ShoppingCartTest client results*

Conclusion

This chapter has covered EJB 3 session bean details using a specific set of examples. We looked at the new and simplified EJB 3 model for developing session beans using standard Java language artifacts, such as Java classes and interfaces. We looked at session beans, and some typical use cases in which session beans can be used for developing applications. We discussed two different types of session beans (stateless and stateful), including the differences between them and some general use cases for each. We covered session bean usage in 2-tier and 3-tier application architectures. We discussed the usage of dependency injection in stateless and stateful beans. We considered ways to gain fine-grained control over application flow, including the use of callback methods and interceptors in stateless and stateful beans, as well as the use of annotations like `@PostContruct` and `@PreDestroy`. We looked at what is required to compile/build, package, and deploy session beans to the GlassFish application server. Finally, we looked at running the sample client programs using the GlassFish application client container.

In the next two chapters, we will drill down into the Java Persistence API (JPA) so that you can learn how to map POJOs to database tables and perform query and CRUD operations.

EJB Fundamentals Enterprise Beans. Â EJB-J2EE Integration: Building a Complete Application The Business Problem A Preview of the Final Web Site Scoping the Technical Requirements. The Business Logic Tier The Presentation Tier Example Code Summary. 606 607 607 608. Â Itâ€™s about EJB concepts, methodology, and development. This book also contains a number of advanced EJB topics, giving you a practical and real-world understanding of the subject. By reading this book, you will acquire a deep understanding of EJB.