Exploring EJB3 With JBoss Application Server
Part -3

By Swaminathan Bhaskar
11/28/2008

5. Stateful Session Bean

A Stateful session bean maintains conversational state between method call invocations for a given client session. When a client does a lookup on a stateful session bean, the EJB container creates an instance of the bean and associates it with that client for the life of the client session. In other words, there is one instance of the stateful session bean per client session. Conversational state is maintained in the instance variable(s) of the stateful session bean. Stateful session beans represent action and are used for modeling business workflow, such as an Online Banking System or Online Shopping System.

We will now see our first stateful session bean in action. Our stateful session bean is a simple Counter bean using EJB3 annotations. Under the directory "**src**" in the Eclipse JbossEJB3, lets create a java package by name "**counter**", which in turn, contains the java packages "**counter.remote**" for the Remote interface, "**counter.bean**" for the Bean implementation, and "**counter.client**" for the EJB client.

The following shows the code for the Remote interface:

```
/*
 * Name: Bhaskar S
 *
 * Date: 11/28/2008
 */

package counter.remote;

import javax.ejb.Remote;

@Remote
public interface Counter {
	public void init(int arg);
	public int next();
	public void destroy();
}
```

The @Remote annotation configures the bean for remote access by the client through this interface.

The following shows the Counter stateful session bean implementation:

```
/*
 * Name: Bhaskar S
 *
 * Date: 11/28/2008
 */

package counter.ejb;

import counter.remote.Counter;

import javax.ejb.Stateful;
```

```
import javax.ejb.Remove;

@Stateful(name="Counter")
public class CounterBean implements Counter {
	private int _cnt;

	@Override
	public void init(int arg) {
		_cnt = arg;
		System.out.println("Called init with " + arg);
	}

	@Override
	public int next() {
		return ++_cnt;
	}

	@Remove
	@Override
	public void destroy() {
		System.out.println("Called destroy");
	}
}
```

The @Stateful annotation configures the bean to be a stateful session bean. The "name" attribute in the @Stateful annotation configures the JNDI name for the EJB. The conversational state in this case is the instance variable "**_cnt**" of type integer. A client will initialize this variable by calling the "**init()**" method. When the client calls "**next()**" each time, the state of the client is maintained by the instance variable "**_cnt**". The @Remove annotation on the method "**destroy()**" indicates to the EJB container that after call to this method, the EJB container can remove the bean instance as the client will no longer be interested in the session.

The following shows a standalone client that is accessing the Counter stateful session bean through the remote interface:

```
/*
 * Name: Bhaskar S
 *
 * Date: 11/28/2008
 */

package counter.client;

import counter.remote.Counter;

import javax.naming.Context;
import javax.naming.InitialContext;

public class CounterClient {
	public static void main(String[] args) {
		if (args.length != 1) {
			System.out.println("Usage: java counter.client.CounterClient <start>");
			System.exit(1);
		}

		try {
			int start = Integer.parseInt(args[0]);

			Context ctx = new InitialContext();

			Counter cntr = (Counter) ctx.lookup("Counter/remote");

			cntr.init(start);

			for (int i = 0; i < 10; i++) {
```

```
                        System.out.println("Next number: " + cntr.next());
                        Thread.sleep(1000);
                    }

                    cntr.destroy();
                }
                catch (Throwable ex) {
                    ex.printStackTrace(System.err);
                }
        }

}
```

Next, we will compile and deploy the EJB jar "**my_ejb_beans.jar**" into the JBoss Application Server. Note that all the EJBs we develop will be bundled into a single jar called "**my_ejb_beans.jar**" for convenience.

Now, open a Terminal window and execute the script "**CounterClient.sh**" as illustrated below:

```
$ CounterClient.sh 100

Next number: 101
Next number: 102
Next number: 103
Next number: 104
Next number: 105
Next number: 106
Next number: 107
Next number: 108
Next number: 109
Next number: 110
```

The above is the output of running the Counter client in a terminal window. The client does a lookup of the Counter stateful session bean. The EJB container creates a new instance of the Counter stateful session bean and assigns it for the client interaction. The client initializes the conversational state of the bean to 100 by calling the method "**init()**". The client next calls the method "**next()**" in a loop 10 times. The client finally terminates after calling the method "**destroy()**". This indicates to the EJB container to remove the session bean instance created for the client.

From the JBoss Application Server log (server/default/log/server.log), we can see the following line when the client calls "**init()**" with 100 on the Counter stateful session bean:

```
19:54:11,669 INFO  [STDOUT] Called init with 100
```

Similarly, we can see the following line when the client class "**destroy()**" on the Counter stateful session bean:

```
19:54:21,746 INFO  [STDOUT] Called destroy
```

With this example, we have successfully deployed and executed our first simple stateful session bean.

For the Counter stateful session bean, we used annotations within regular Java classes to make it behave like an EJB. Rather than using annotations, we could use an XML deployment descriptor file to configure regular Java classes. Lets look at a prepaid phone card stateful session bean which uses XML configuration file rather than annotations.

The following shows the code for a regular Java interface PhoneCard:

```
/*
 * Name: Bhaskar S
 *
 * Date: 11/28/2008
 */

package prepaid.remote;

public interface PhoneCard {
	public void setup(int arg);
	public int talk(int arg);
	public void discard();
}
```

The following shows a regular Java class that implements the interface PhoneCard:

```
/*
 * Name: Bhaskar S
 *
 * Date: 11/28/2008
 */

package prepaid.ejb;

import prepaid.remote.PhoneCard;

public class PhoneCardBean implements PhoneCard {
	private int _mins;

	@Override
	public void setup(int arg) {
		_mins = arg;

		System.out.println("Phone card has " + _mins + " minutes");
	}

	@Override
	public int talk(int arg) {
		if (arg < _mins) {
			_mins -= arg;
		}
		else {
			_mins = 0;
		}
		return _mins;
	}

	@Override
	public void discard() {
		System.out.println("Phone card is used !!!");
	}
}
```

The following shows the XML deployment descriptor file "**ejb-jar.xml**" that configures the above indicated regular Java classes as a stateful session bean:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
       version="3.0">
    <description>My EJBs</description>
    <display-name>My EJBs</display-name>
       <enterprise-beans>
        <session>
            <ejb-name>PhoneCard</ejb-name>
            <remote>prepaid.remote.PhoneCard</remote>
            <ejb-class>prepaid.ejb.PhoneCardBean</ejb-class>
            <session-type>Stateful</session-type>
            <remove-method>
                <bean-method>
                    <method-name>discard</method-name>
                </bean-method>
            </remove-method>
            <transaction-type>Container</transaction-type>
        </session>
       </enterprise-beans>
</ejb-jar>
```

For brevity sake, we are ignoring the other EJBs that were configured in this XML file from **Part-2** of this series.

From the above configuration file, it is clear that the EJB is named "**PhoneCard**" and it is a stateful session bean. The configuration also indicates the remote interface and the bean implementation class. In addition, the configuration indicates the remove method to be "**discard()**".

The following shows a standalone client that is accessing the PhoneCard stateful session bean through the remote interface:

```
/*
 * Name: Bhaskar S
 *
 * Date: 11/28/2008
 */

package prepaid.client;

import prepaid.remote.PhoneCard;

import javax.naming.Context;
import javax.naming.InitialContext;

public class PhoneCardClient {
       public static void main(String[] args) {
              if (args.length != 1) {
                     System.out.println("Usage: java prepaid.client.PhoneCardClient <minutes>");
                     System.exit(1);
              }

              try {
                     int mins = Integer.parseInt(args[0]);

                     Context ctx = new InitialContext();

                     PhoneCard card = (PhoneCard) ctx.lookup("PhoneCard/remote");
                     card.setup(mins);
                     while (true) {
                            int remaining = card.talk(10);
```

```
                        System.out.println("Remaining minutes: " + remaining);
                        if (remaining == 0) {
                                break;
                        }
                    }
                    card.discard();
                }
                catch (Throwable ex) {
                    ex.printStackTrace(System.err);
                }
        }
}
```

Next, we will compile and deploy the EJB jar "**my_ejb_beans.jar**" into the JBoss Application Server.

Now, open a Terminal window and execute the script "**PhoneCardClient.sh**" as illustrated below:

```
$ PhoneCardClient.sh 100

Remaining minutes: 90
Remaining minutes: 80
Remaining minutes: 70
Remaining minutes: 60
Remaining minutes: 50
Remaining minutes: 40
Remaining minutes: 30
Remaining minutes: 20
Remaining minutes: 10
Remaining minutes: 0
```

The above is the output of running the PhoneCard client in a terminal window. The client does a lookup of the PhoneCard stateful session bean. The client initializes the conversational state of the bean, which is the phone card minutes in the case, to 100 minutes by calling the method "**setup()**". The client next calls the method "**talk()**" in a loop till the phone card minutes are exhausted. The client finally terminates after calling the method "**discard()**".

The life cycle of Stateful session beans are a little more complicated compared to stateless session beans. Stateful session beans maintain client state and hence associates one bean instance for each client instance. This can strain system resources on the EJB container when there are large number of concurrent client requests. To handle large number of concurrent clients, the EJB container may decide to temporarily swap the state of one or more of the idle bean instances to the secondary store such as the filesystem. This process is called **Passivation**. The reverse process of bring a temporarily swapped bean instance back from the secondary store to the EJB container is called **Activation**.

A stateful session bean can have internal references to system resources such as a database connection object or a socket connection object, etc. That said, a stateful session bean must release references to system resources on Passivation and acquire them on Activation. The EJB3 specification supports the following life cycle callback methods for managing references to system resources:

Post-Construct	Called after a new instance of stateful session bean is created
Pre-Destroy	Called before a stateful session bean instance is to be removed

Pre-Passivate	Called before a stateful session bean instance is to be passivated
Post-Activate	Called after a stateful session bean instance is activated

When a client does a lookup on a stateful session bean, it is said to be in the "**Does Not Exist**" state. To handle the client request, the EJB container creates a new instance of the bean, injects all the necessary dependencies into the bean instance, and finally invokes the Post-Construct method, if one is defined. Now, the bean instance enters the "**Method Ready**" state and is used to serve the client request. When there are a large number of active bean instances and the EJB container has to passivate an idle bean instance to conserve resources, it will invoke the Pre-Passivate method, if there is one defined for the bean. Later, when a bean has to be activated, the EJB container will activate the bean from the secondary store and then invoke the Post-Activate method, if there is one defined for the bean. Finally, when the client session terminates, the EJB container will remove the bean instance. But, before removing a bean instance, it invokes the Pre-Destroy method, if one is defined.

In the following example, we will demonstrate the functionality of life cycle callback methods using a hypothetical clock service. In this example, we will use annotations to configure the Pre-Passivate, Post-Activate, Post-Construct and Pre-Destroy life cycle callback methods of the Stateful session bean.

The following shows the code for the Remote interface CustomClock:

```
/*
 * Name: Bhaskar S
 *
 * Date: 11/28/2008
 */

package clock.remote;

import javax.ejb.Remote;

@Remote
public interface CustomClock {
	public void init(String name);
	public String getTime();
	public void destroy();
}
```

The following shows the CustomClock stateful session bean implementation:

```
/*
 * Name: Bhaskar S
 *
 * Date: 11/28/2008
 */

package clock.ejb;

import clock.remote.CustomClock;

import java.util.Calendar;
import java.util.GregorianCalendar;

import javax.ejb.Stateful;
import javax.ejb.PrePassivate;
import javax.ejb.PostActivate;
import javax.ejb.Remove;
import javax.annotation.PostConstruct;
```

```
import javax.annotation.PreDestroy;

@Stateful(name="CustomClock")
public class CustomClockBean implements CustomClock {
	private String _name;
	private Calendar _cal;

	@Override
	public void init(String name) {
		_name = name;
	}

	@Override
	public String getTime() {
		StringBuffer sb = new StringBuffer();
		sb.append("Hi ");
		sb.append(_name);
		sb.append(", the time now is ");
		sb.append(_cal.get(Calendar.HOUR));
		sb.append(":");
		sb.append(_cal.get(Calendar.MINUTE));
		sb.append(":");
		sb.append(_cal.get(Calendar.SECOND));
		sb.append(":");
		int ap = _cal.get(Calendar.AM_PM);
		if (ap == 0) {
			sb.append(" AM");
		}
		else if (ap == 1) {
			sb.append(" PM");
		}
		return sb.toString();
	}

	@Remove
	@Override
	public void destroy() {
	}

	@PrePassivate
	public void prePassivate() {
		_cal = null;

		System.out.println("prePassivate(): pre-passivate life cycle method callback");
	}

	@PostActivate
	public void postActivate() {
		_cal = new GregorianCalendar();

		System.out.println("postActivate(): post-activate life cycle method callback");
	}

	@PostConstruct
	public void postConstruct() {
		_cal = new GregorianCalendar();

		System.out.println("postConstruct(): post-construct life cycle method callback");
	}

	@PreDestroy
	public void preDestroy() {
		_cal = null;

		System.out.println("preDestroy(): pre-destroy life cycle method callback");
	}
}
```

The @PrePassivate annotation indicates that the method "**prePassivate()**" will be invoked just before the CustomClockBean is to be passivated. The @PrePassivate annotation can be applied only to a no-argument "**public void**" method.

The @PostActivate annotation indicates that the method "**postActivate()**" will be invoked after the CustomClockBean instance is activated from the secondary store. The @PostActivate annotation can be applied only to a no-argument "**public void**" method.

In order to test the functionality of Passivation and Activation, we will need a JBoss Application Server specific XML configuration called "**jboss.xml**" which will reside in the "**META-INF**" directory and is shown as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss>
    <enterprise-beans>
        <session>
            <ejb-name>CustomClock</ejb-name>
            <cache-config>
                <idle-timeout-seconds>3</idle-timeout-seconds>
            </cache-config>
        </session>
    </enterprise-beans>
</jboss>
```

In the above JBoss Application Server specific XML configuration, we set the idle timeout (in seconds) for the CustomClock stateful session bean to 3 seconds. After the stateful session bean has been idle for this time period, it will be automatically passivated by the JBoss Application Server. Accessing a method on an already passivated stateful session bean will activate it.

The following shows a standalone client that is accessing the CustomClock stateful session bean through the remote interface:

```
/*
 * Name: Bhaskar S
 *
 * Date: 11/28/2008
 */

package clock.client;

import clock.remote.CustomClock;

import java.io.BufferedReader;
import java.io.InputStreamReader;

import javax.naming.Context;
import javax.naming.InitialContext;

public class CustomClockClient {
       public static void main(String[] args) {
              if (args.length != 1) {
                     System.out.println("Usage: java clock.client.CustomClockClient <name>");
                     System.exit(1);
              }

              try {
                     BufferedReader input = new BufferedReader(new InputStreamReader(System.in));

                     Context ctx = new InitialContext();

                     CustomClock clck = (CustomClock) ctx.lookup("CustomClock/remote");
                     clck.init(args[0]);
                     boolean exit = false;
                     while (! exit) {
```

```
                                System.out.println(clck.getTime());
                                System.out.print("Type q and <Enter> to quit: ");
                                String prompt = input.readLine();
                                if (prompt.equalsIgnoreCase("q")) {
                                        exit = true;
                                }
                        }
                        clck.destroy();
                }
                catch (Throwable ex) {
                        ex.printStackTrace(System.err);
                }
        }
}
```

Next, we will compile and deploy the EJB jar "**my_ejb_beans.jar**" into the JBoss Application Server.

Now, open a Terminal window and execute the script "**CustomClockClient.sh**" as illustrated below:

```
$ CustomClockClient.sh Bhaskar

Hi Bhaskar, the time now is 9:10:59: PM
Type q and <Enter> to quit: <Enter>
Hi Bhaskar, the time now is 9:11:10: PM
Type q and <Enter> to quit: <Enter>
Hi Bhaskar, the time now is 9:11:10: PM
Type q and <Enter> to quit: <Enter>
Hi Bhaskar, the time now is 9:11:22: PM
Type q and <Enter> to quit: q
```

The following output shows the log lines from the JBoss Application Server log (server/default/log/server.log):

```
21:10:59,452 INFO  [STDOUT] postConstruct(): post-construct life cycle method callback
21:11:03,764 INFO  [STDOUT] prePassivate(): pre-passivate life cycle method callback for Bhaskar
21:11:10,365 INFO  [STDOUT] postActivate(): post-activate life cycle method callback for Bhaskar
21:11:15,766 INFO  [STDOUT] prePassivate(): pre-passivate life cycle method callback for Bhaskar
21:11:22,240 INFO  [STDOUT] postActivate(): post-activate life cycle method callback for Bhaskar
21:11:22,948 INFO  [STDOUT] preDestroy(): pre-destroy life cycle method callback for Bhaskar
```

From the above, it is clear that when the client waits more than 3 seconds (idle timeout is 3 seconds) to press <Enter>, the CustomClockBean times out due to inactivity and the EJB container passivates the bean instance to the secondary store. When the client presses <Enter> and calls "getTime()", the bean instance is activated and the current time is returned to the client.

I leave it to the readers to try out the same example without annotations. Have some fun !!!

Earlier we indicated that Stateful session beans are used to model workflow process. One could ask the question why should Stateful session beans model workflow process ? Let us answer that question by considering the example of a hypothetical online Bill Payment system. At the very least, we will need a Customer Account bean and a Payment Processor bean to pay bills. The following are the steps to pay a bill to Company XYZ for $250.00:

a. Debit the Customer Account for $250.00. If the account has insufficient cash, throw an exception

b. Pay the bill to Company XYZ for $250.00
c. Update the Customer Account of the payment transaction

If we don't use a Stateful session bean, the client code will have perform all these steps. This implies the logic for paying bills now permeates the client code and any changes to the steps or any of the beans affects the client code due to the tight coupling. This is not elegant.

Also, when the client invokes a method on the remote interface of the bean, data is actually serialized and shipped to the EJB container where actual bean instance resides. When the bean method completes, any return value is serialized and sent back to the client. Considering the above example, the client code will have to incur a lot of network round-trips to pay a bill of $250 to Company XYZ. This is very inefficient.

By using a Stateful session bean, we could have encapsulated the steps to pay bills in the bean and presented a very simple interface to the client, which would have been more elegant and efficient. This is often referred to as a "**Session Facade**" in the design patterns lingo.

In the following example, we will explore the hypothetical online Bill Payment system. We are going to keep this example very simple without involving the database.

The following shows the code for the custom exception thrown when there is insufficient cash in the customer account to pay the bill:

```
/*
 * Name: Bhaskar S
 *
 * Date: 11/28/2008
 */

package bank.common;

public class InsufficientCashException extends Exception {
	private static final long serialVersionUID = 1L;

	public InsufficientCashException() {
		super();
	}

	public InsufficientCashException(String msg) {
		super(msg);
	}
}
```

The following shows the code for the object that represents a bill payment transaction:

```
/*
 * Name: Bhaskar S
 *
 * Date: 11/28/2008
 */

package bank.common;

import java.io.Serializable;
import java.util.Date;

public class BillPayment implements Serializable {
```

```
	private static final long serialVersionUID = 1L;

	private double _amount;
	private String _entity;
	private Date _date;

	public BillPayment(String ent, double amt) {
		_entity = ent;
		_amount = amt;
		_date = new Date();
	}

	public String toString() {
		return "-> Date: " + _date + ", Entity: " + _entity + ", Amount: " + _amount;
	}
}
```

The following shows the code for a local interface for Customer Account:

```
/*
 * Name: Bhaskar S
 *
 * Date: 11/28/2008
 */

package bank.local;

import javax.ejb.Local;

import bank.common.InsufficientCashException;
import bank.common.BillPayment;

@Local
public interface CustomerAccount {
	public void init(String arg);
	public double credit(double amt);
	public double debit(double amt)
		throws InsufficientCashException;
	public void addBillPayment(BillPayment txn);
	public String showAccountDetails();
}
```

The @Local annotation configures the bean for local access within the EJB container. This means any other bean may communicate with it locally more efficiently. This bean cannot be accessed by external client for remote access.

The following shows a Stateful session bean that implements the interface CustomerAccount:

```
/*
 * Name: Bhaskar S
 *
 * Date: 11/28/2008
 */

package bank.ejb;

import java.util.ArrayList;

import javax.ejb.Stateful;

import bank.common.InsufficientCashException;
import bank.common.BillPayment;
import bank.local.CustomerAccount;
```

```
@Stateful(name="CustomerAccount")
public class CustomerAccountBean implements CustomerAccount {
	private double _cash = 0.0;
	private String _name;
	private ArrayList<BillPayment> _list = new ArrayList<BillPayment>();

	@Override
	public void init(String arg) {
		_name = arg;
	}

	@Override
	public double credit(double amt) {
		if (amt > 0.0) {
			_cash += amt;
		}
		return _cash;
	}

	@Override
	public double debit(double amt)
		throws InsufficientCashException {
		if (amt > 0.0) {
			if (_cash < amt) {
				throw new InsufficientCashException("Insufficient cash in account,
balance: " + _cash);
			}
			_cash -= amt;
		}
		return _cash;
	}

	@Override
	public void addBillPayment(BillPayment txn) {
		if (txn != null) {
			_list.add(txn);
		}
	}

	@Override
	public String showAccountDetails() {
		StringBuffer sb = new StringBuffer();
		sb.append("Name: ");
		sb.append(_name);
		sb.append("\n");
		sb.append("Cash: ");
		sb.append(_cash);
		sb.append("\n");
		sb.append("Bill Payment Transaction(s):\n");
		for (BillPayment txn : _list) {
			sb.append("\t");
			sb.append(txn.toString());
			sb.append("\n");
		}
		return sb.toString();
	}
}
```

There needs to be one CustomerAccount bean instance per customer account and hence it is modeled as a stateful session bean.

The CustomerAccount bean supports the following operations:

1. Credit cash into the account
2. Debit cash from the account to pay bill
3. Add a bill payment transaction
4. Display the current state of the account including bill payment transactions

The following shows the code for another local interface for Payment Processor:

```
/*
 * Name: Bhaskar S
 *
 * Date: 11/28/2008
 */

package bank.local;

import javax.ejb.Local;

import bank.common.BillPayment;

@Local
public interface PaymentProcessor {
	public BillPayment pay(String ent, double amt);
}
```

The following shows a Stateless session bean that implements the interface PaymentProcessor:

```
/*
 * Name: Bhaskar S
 *
 * Date: 11/28/2008
 */

package bank.ejb;

import javax.ejb.Stateless;

import bank.common.BillPayment;
import bank.local.PaymentProcessor;

@Stateless(name="PaymentProcessor")
public class PaymentProcessorBean implements PaymentProcessor {
	@Override
	public BillPayment pay(String ent, double amt) {
		System.out.println("+++++ Bill paid for entity: " + ent + " for amount: " + amt);
		return new BillPayment(ent, amt);
	}
}
```

The PaymentProcessor bean is modeled as a stateless session bean since it can be used by any customer to pay bills.

The PaymentProcessor bean supports only one operation: Pay a customers bill. It returns a bill payment transaction object called BillPayment.

The following shows the remote interface Bank:

```
/*
 * Name: Bhaskar S
 *
 * Date: 11/28/2008
 */

package bank.remote;

import javax.ejb.Remote;
```

```
import bank.common.InsufficientCashException;

@Remote
public interface Bank {
	public void init(String arg);
	public double credit(double amt);
	public void payBill(String ent, double amt)
		throws InsufficientCashException;
	public String showAccountDetails();
}
```

This is the interface that is exposed to the remote client. In other words, Bank is the "**Session Facade**".

The following shows a Stateful session bean that implements the interface Bank:

```
/*
 * Name: Bhaskar S
 *
 * Date: 11/28/2008
 */

package bank.ejb;

import javax.ejb.Stateful;
import javax.ejb.EJB;

import bank.common.InsufficientCashException;
import bank.local.CustomerAccount;
import bank.local.PaymentProcessor;
import bank.remote.Bank;

@Stateful(name="Bank")
public class BankBean implements Bank {
	@EJB private CustomerAccount _custAcct;
	@EJB private PaymentProcessor _payProc;

	@Override
	public void init(String arg) {
		_custAcct.init(arg);
	}

	@Override
	public double credit(double amt) {
		return _custAcct.credit(amt);
	}

	@Override
	public void payBill(String ent, double amt)
			throws InsufficientCashException {
		_custAcct.debit(amt);
		_custAcct.addBillPayment(_payProc.pay(ent, amt));
	}

	@Override
	public String showAccountDetails() {
		return _custAcct.showAccountDetails();
	}
}
```

The @EJB annotation indicates to the EJB container that it should lookup and inject an EJB that implements some interface. In the above example, the EJB container will inject two EJBs: one EJB of type CustomerAccount and another EJB of type PaymentProcessor. The injected instances are not the actual bean instances. Instead, it is a local proxy that implements the appropriate interfaces.

This stateful session bean acts as a "**Session Facade**" for a client providing the necessary services for the customers. This stateful session beans internally uses local EJBs to perform the necessary operations on behalf of the customers.

The following shows a standalone client that is accessing the Bank stateful session bean through the remote interface:

```
/*
 * Name: Bhaskar S
 *
 * Date: 11/28/2008
 */

package bank.client;

import java.io.BufferedReader;
import java.io.InputStreamReader;

import javax.naming.Context;
import javax.naming.InitialContext;

import bank.remote.Bank;

public class BankClient {
	public static void main(String[] args) {
		if (args.length != 1) {
			System.out.println("Usage: java bank.client.BankClient <name>");
			System.exit(1);
		}

		try {
			BufferedReader input = new BufferedReader(new InputStreamReader(System.in));

			Context ctx = new InitialContext();

			Bank bank = (Bank) ctx.lookup("Bank/remote");
			bank.init(args[0]);

			boolean exit = false;
			while (! exit) {
				System.out.println("1. Credit");
				System.out.println("2. Pay Bill");
				System.out.println("3. Account Details");
				System.out.println("4. Quit");

				String choice = input.readLine();

				if (choice.equals("1")) {
					System.out.print("Amount: ");
					String amtStr = input.readLine();
					double amt = bank.credit(Double.parseDouble(amtStr));
					System.out.println("-> Cash balance after credit: " + amt);
				}
				else if (choice.equals("2")) {
					System.out.print("Entity: ");
					String entStr = input.readLine();
					System.out.print("Amount: ");
					String amtStr = input.readLine();
					try {
						bank.payBill(entStr, Double.parseDouble(amtStr));
					}
					catch (Exception e) {
						System.err.println("*** Error: " + e.getMessage());
					}
				}
				else if (choice.equals("3")) {
					System.out.println(bank.showAccountDetails());
				}
				else if (choice.equals("4")) {
```

```
                                        exit = true;
                                }
                        }
                }
                catch (Throwable ex) {
                        ex.printStackTrace(System.err);
                }
        }
}
```

We have made this client as an interactive client to simulate the various customer requests. It prompts for choices and user responses and calls the appropriate operations on the Bank interface.

Next, we will compile and deploy the EJB jar "**my_ejb_beans.jar**" into the JBoss Application Server.

Now, open a Terminal window and execute the script "**BankClient.sh**" as illustrated below:

```
$ BankClient.sh Bhaskar

1. Credit
2. Pay Bill
3. Account Details
4. Quit
1
Amount: 100.50
-> Cash balance after credit: 100.5
1. Credit
2. Pay Bill
3. Account Details
4. Quit
2
Entity: Company XYZ
Amount: 72.25
1. Credit
2. Pay Bill
3. Account Details
4. Quit
2
Entity: Firm ABC
Amount: 33.67
*** Error: Insufficient cash in account, balance: 28.25
1. Credit
2. Pay Bill
3. Account Details
4. Quit
3
Name: Bhaskar
Cash: 28.25
Bill Payment Transaction(s):
        -> Date: Sat Dec 06 00:14:20 EST 2008, Entity: Company XYZ, Amount: 72.25
```

```
1. Credit
2. Pay Bill
3. Account Details
4. Quit
4
```

Want to try the above example with XML deployment descriptors ? Well then I leave that exercise to the readers to explore. (**Hint**: I have the example in the XML deployment descriptor “ejb-jar.xml” as Bank2)

This concludes our journey into Stateful Session Beans.

We will explore Message Driven Beans in Part-4 of this series.