Exploring EJB3 With JBoss Application Server
Part - 5

By Swaminathan Bhaskar
12/13/2008

In this part, we will first explore EJB Timer Service and then look at Interceptors.

7. EJB Timer Service

A Timer Service is a EJB container managed scheduler component that allows a stateless session bean or message driven bean to register with it for a callback at a specified time or after some elapsed time or at recurring time intervals. This time-based notification service allows one to build routine scheduled jobs such as generating reports at end of business day or DB maintenance, etc.

To register with the Timer Service, a stateless session bean or a message driven bean will perform the following steps:

- Use dependency injection to get a reference to an instance of the Timer Service
- Use the handle to the Timer Service to create a Timer that will expire at the appropriate time
- Provide a Timeout callback method which the Timer Service will callback into on Timer expiry

We will now look at a simple example that will use the Timer Service. Consider the example of an Online Toy Shop. The Toy Shop management wants a report at regular intervals to see which of the Toys are popular and selling more.

Before we can proceed, we need to do some modifications to the standard JBoss Application Server. By default, a Timer is persistent. When we create a Timer using the Timer Service, it is persisted in a secondary store such as a database. This means that a Timer can survive system crashes. If we create a Timer and the system crashes and later when the JBoss Application Server comes back online, the persisted Timer is reactivated. For our example, we do not need a persistent Timer. To disable persistent Timer in JBoss, we need to make the following changes to the configuration file "**ejb-deployer.xml**" located under $JBOSS_HOME/server/default/deploy:

Comment the mbean that looks as shown below:

```
<mbean code="org.jboss.ejb.txtimer.DatabasePersistencePolicy"
          name="jboss.ejb:service=EJBTimerService,persistencePolicy=database">
```

Once you comment it out, it should look as follows:

```
 <!-- Disable Timer persistence -->
 <!--
 <mbean code="org.jboss.ejb.txtimer.DatabasePersistencePolicy"
       name="jboss.ejb:service=EJBTimerService,persistencePolicy=database">
  <depends optional-attribute-name="DataSource">
    jboss.jca:service=DataSourceBinding,name=DefaultDS
```

```
   </depends>
   <attribute name="DatabasePersistencePlugin">
      org.jboss.ejb.txtimer.GeneralPurposeDatabasePersistencePlugin
   </attribute>
   <attribute name="TimersTable">TIMERS</attribute>
  </mbean>
  -->
```

Next, uncomment the following:

```
  <!-- A persistence policy that does not persist the timers
  <mbean code="org.jboss.ejb.txtimer.NoopPersistencePolicy"
        name="jboss.ejb:service=EJBTimerService,persistencePolicy=noop"/>
  -->
```

After uncommenting, it should look as follows:

```
  <mbean code="org.jboss.ejb.txtimer.NoopPersistencePolicy"
        name="jboss.ejb:service=EJBTimerService,persistencePolicy=noop"/>
```

With this we have now configured the JBoss Timer Service to be non-persistent.

Next, we need to setup a table in the MySQL database called "**testdb**". Create a table called "**TOY_SALE_TBL**" as follows:

```
mysql> CREATE TABLE TOY_SALE_TBL ( ORDER_NO INT NOT NULL AUTO_INCREMENT, TOY_SOLD VARCHAR(30) NULL, QUANTITY SMALLINT NULL, PRIMARY KEY ORDER_NO_PK (ORDER_NO) );

mysql> CREATE INDEX TOY_SOLD_IDX ON TOY_SALE_TBL (TOY_SOLD);
```

We will need the above table for our Online Toy Store example. Now we are ready to proceed.

The following shows the code for the Remote interface called "**ShopToy**":

```
/*
 * Name: Bhaskar S
 *
 * Date: 12/14/2008
 */

package shop.remote;

import javax.ejb.Remote;

@Remote
public interface ShopToy {
       public void buy(String toy, int qty);
}
```

The following shows the code for the Remote interface called "**ShopToyScheduler**":

```
/*
 * Name: Bhaskar S
 *
 * Date: 12/14/2008
 */

package shop.remote;

import javax.ejb.Remote;

@Remote
public interface ShopToyScheduler {
	public void createTimer();
	public void cancelTimer();
}
```

Notice that we have two remote interfaces. The stateless session bean will implement both these interfaces. Just bear with me – will explain in a little bit.

The following shows the **ShopToyBean** stateless session bean implementation:

```
/*
 * Name: Bhaskar S
 *
 * Date: 12/14/2008
 */

package shop.ejb;

import shop.remote.ShopToy;
import shop.remote.ShopToyScheduler;

import javax.ejb.Stateless;
import javax.ejb.Timeout;
import javax.ejb.Timer;
import javax.ejb.TimerService;
import javax.annotation.Resource;

import javax.sql.DataSource;

import java.sql.Connection;
import java.sql.Statement;
import java.sql.ResultSet;

@Stateless(name="ShopToy")
public class ShopToyBean implements ShopToy, ShopToyScheduler {
	@Resource(mappedName="java:/MySqlDS") private DataSource _dataSource;
	@Resource TimerService _timerService; // 1

	@Override
	public void buy(String toy, int qty) {
		Connection con = null;
		Statement stmt = null;
		try {
			con = _dataSource.getConnection();
			stmt = con.createStatement();
			stmt.executeUpdate("INSERT INTO TOY_SALE_TBL (TOY_SOLD, QUANTITY) VALUES ('" +
toy + "', " + qty + ")");
		}
		catch (Exception ex) {
			throw new RuntimeException(ex.getCause());
		}
		finally {
```

```
                        closeStatement(stmt);
                        closeConnection(con);
                }
        }

        @Timeout
        public void timerCallback(Timer timer) { // 2
                Connection con = null;
                Statement stmt = null;
                ResultSet rs = null;
                try {
                        con = _dataSource.getConnection();
                        stmt = con.createStatement();
                        rs = stmt.executeQuery("SELECT TOY_SOLD, sum(QUANTITY) FROM TOY_SALE_TBL GROUP
BY TOY_SOLD");
                        System.out.println("Report of Popular Toys Sold");
                        System.out.println("---------------------------");
                        int i = 0;
                        while (rs.next()) {
                                i++;
                                System.out.println(i + ". Toy: " + rs.getString(1) + ", Quantity Sold: "
+ rs.getInt(2));
                        }
                }
                catch (Exception ex) {
                        throw new RuntimeException(ex.getCause());
                }
                finally {
                        closeResultSet(rs);
                        closeStatement(stmt);
                        closeConnection(con);
                }
        }

        @Override
        public void createTimer() {
                // Recurring timer - every 5 mins
                _timerService.createTimer(300000, 300000, "ShopToy"); //3
        }

        @Override
        public void cancelTimer() {
                for (Object obj: _timerService.getTimers()) {
                        Timer timer = (Timer) obj;
                        if ("ShopToy".equals((String)timer.getInfo())) { // 4
                                timer.cancel();
                        }
                }
        }

        void closeResultSet(ResultSet rs) {
                if (rs != null) {
                        try {
                                rs.close();
                        }
                        catch (Exception e) {
                        }
                }
        }

        void closeStatement(Statement stmt) {
                if (stmt != null) {
                        try {
                                stmt.close();
                        }
                        catch (Exception e) {
                        }
                }
        }

        void closeConnection(Connection con) {
                if (con != null) {
                        try {
                                con.close();
```

```
                    }
                    catch (Exception e) {
                    }
                }
        }
}
```

The following will explain the highlighted steps from the code:

1. We are using Dependency Injection to initialize the handle to the TimerService object. The @Resource annotation causes the EJB container to inject the handle to the TimerService to the variable named _timerService
2. The @Timeout annotation configures the method "**timerCallback**" to be the callback on Timer expiry. The callback method passes a parameter of type Timer, which is passed by the TimerService on callback. When a timer expires, the EJB container will select a bean instance from the pool and call the timeout method which is "**timerCallback**" in this case
3. The TimerService has number of methods to create the appropriate Timer. We have chosen the one with the method signature: **public Timer createTimer(long initialDuration, long duration, Serializable info)**. This method creates a recurring timer. The first parameter is the initial duration before the timer expires in milliseconds. In our case it is five minutes (300000 milliseconds). The second parameter is the repeating duration. In our case it is five minutes (300000 milliseconds). Because of the second parameter, our timer will fire every five minutes. The third parameter is handle to a context object which is the string "**ShopToy**" in our case
4. The TimerService will allows us to iterate through all the Timers registered with it. Since we should only be canceling the Timer created for "**ShopToy**", we first check the context information to see if it is "**ShopToy**" (same as the third parameter when we created the Timer). If true, we cancel the Timer.

So, in our example, why does the bean implement two interfaces ? What we have done is that we have separated the business methods from the timer management methods. So, why have timer management methods ? Since its a stateless session beans, the EJB container will have a pool of the bean instances. As a result, we cannot create the timer in either the constructor or the Post-Construct method as it will create the timer for each instance in pool. This will cause the timeout method to be called multiple times which is not the desired behavior. We want the timer to be created once and hence have a separate interface for the timer management.

The following shows a standalone client that is accessing the ShopToy stateless session bean through the remote interface "**ShopToyScheduler**":

```
/*
 * Name: Bhaskar S
 *
 * Date: 12/14/2008
 */

package shop.client;

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

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

import shop.remote.ShopToyScheduler;
```

```
public class ShopToyAdmin {
	public static void main(String[] args) {
		try {
			BufferedReader input = new BufferedReader(new InputStreamReader(System.in));

			Context ctx = new InitialContext();

			ShopToyScheduler admin = (ShopToyScheduler) ctx.lookup("ShopToy/remote");

			boolean exit = false;
			while (! exit) {
				System.out.println("1. Create Timer");
				System.out.println("2. Cancel Timer");
				System.out.println("3. Quit");

				String choice = input.readLine();

				if (choice.equals("1")) {
					admin.createTimer();
					System.out.println("-> Created ShopToy Timer");
				}
				else if (choice.equals("2")) {
					admin.cancelTimer();
					System.out.println("-> Cancelled ShpToy Timer");
				}
				else if (choice.equals("3")) {
					exit = true;
				}
			}
		}
		catch (Throwable ex) {
			ex.printStackTrace(System.err);
		}
	}
}
```

The above client is used for managing the EJB Timer for the stateless session bean “ShopToy”.

The following shows a standalone client that is accessing the ShopToy stateless session bean through the remote interface “**ShopToy**”:

```
/*
 * Name: Bhaskar S
 *
 * Date: 12/14/2008
 */

package shop.client;

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

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

import shop.remote.ShopToy;

public class ShopToyClient {
	public static void main(String[] args) {
		try {
			BufferedReader input = new BufferedReader(new InputStreamReader(System.in));

			Context ctx = new InitialContext();

			ShopToy shop = (ShopToy) ctx.lookup("ShopToy/remote");

			boolean exit = false;
```

```
                    while (! exit) {
                            System.out.println("1. Buy Toy ABC");
                            System.out.println("2. Buy Toy PQR");
                            System.out.println("3. Buy Toy XYZ");
                            System.out.println("4. Quit");

                            String choice = input.readLine();

                            if (choice.equals("1")) {
                                    System.out.print("Quantity: ");
                                    String qtyStr = input.readLine();
                                    int qty = Integer.parseInt(qtyStr);
                                    shop.buy("ABC", qty);
                                    System.out.println("-> Bought " + qty + " of ABC");
                            }
                            else if (choice.equals("2")) {
                                    System.out.print("Quantity: ");
                                    String qtyStr = input.readLine();
                                    int qty = Integer.parseInt(qtyStr);
                                    shop.buy("PQR", qty);
                                    System.out.println("-> Bought " + qty + " of PQR");
                            }
                            else if (choice.equals("3")) {
                                    System.out.print("Quantity: ");
                                    String qtyStr = input.readLine();
                                    int qty = Integer.parseInt(qtyStr);
                                    shop.buy("XYZ", qty);
                                    System.out.println("-> Bought " + qty + " of XYZ");
                            }
                            else if (choice.equals("4")) {
                                    exit = true;
                            }
                    }
            }
            catch (Throwable ex) {
                    ex.printStackTrace(System.err);
            }
    }
}
```

The above client represents the online toy shopping experience.

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 (Term-1) and execute the script "**ShopToyAdmin.sh**" as illustrated below:

```
$ ShopToyAdmin.sh

1. Create Timer
2. Cancel Timer
3. Quit
1
-> Created ShopToy Timer
```

By choosing option 1 from the menu choices, we are creating and registering a recurring Timer with the TimerService in the EJB container.

Now, open one other Terminal window (Term-2) and execute the script "**ShopToyClient.sh**" as

illustrated below:

```
$ ShopToy.sh

1. Buy Toy ABC
2. Buy Toy PQR
3. Buy Toy XYZ
4. Quit
1
Quantity: 1
-> Bought 1 of ABC
1. Buy Toy ABC
2. Buy Toy PQR
3. Buy Toy XYZ
4. Quit
2
Quantity: 2
-> Bought 2 of PQR
1. Buy Toy ABC
2. Buy Toy PQR
3. Buy Toy XYZ
4. Quit
3
Quantity: 5
-> Bought 5 of XYZ
```

The above represents the toy purchases made by a client.

From the JBoss Application Server log (server/default/log/server.log), we can see the following line when the the initial timer fires after 5 mins:

```
21:43:47,989 INFO  [STDOUT] Report of Popular Toys Sold
21:43:47,989 INFO  [STDOUT] --------------------------
21:43:47,989 INFO  [STDOUT] 1. Toy: ABC, Quantity Sold: 1
21:43:47,989 INFO  [STDOUT] 2. Toy: PQR, Quantity Sold: 2
21:43:47,989 INFO  [STDOUT] 3. Toy: XYZ, Quantity Sold: 5
```

Make more purchases in the Terminal window (Term-2) as shown below:

```
1. Buy Toy ABC
2. Buy Toy PQR
3. Buy Toy XYZ
4. Quit
1
Quantity: 3
-> Bought 3 of ABC
1. Buy Toy ABC
2. Buy Toy PQR
3. Buy Toy XYZ
```

```
4. Quit
4
```

Next time the recurring timer fires, we will see the following in the JBoss Application Server log (server/default/log/server.log):

```
21:48:47,980 INFO  [STDOUT] Report of Popular Toys Sold
21:48:47,980 INFO  [STDOUT] --------------------------
21:48:47,980 INFO  [STDOUT] 1. Toy: ABC, Quantity Sold: 4
21:48:47,981 INFO  [STDOUT] 2. Toy: PQR, Quantity Sold: 2
21:48:47,981 INFO  [STDOUT] 3. Toy: XYZ, Quantity Sold: 5
```

Now, we will cancel the timer in Terminal window (Term-1) as shown below:

```
1. Create Timer
2. Cancel Timer
3. Quit
2
-> Cancelled ShpToy Timer
1. Create Timer
2. Cancel Timer
3. Quit
3
```

After this we will see no more reports being generated as there are no more timeouts.

I leave it to the readers to try out the same example using the XML deployment descriptor as an exercise.

8. Interceptors

Interceptors are a rudimentary form of Aspect Oriented Programming at Method level. What is Aspect Oriented Programming ? Imagine you want to find how long it takes to execute a method called "**methodOne()**". This entails one to modify the code for "**methodOne()**" as follows:

```
.
.
.
public void methodOne() {
    long start = System.currentTimeMillis();
    try {
        ...
    }
    catch (Exception ex) {
        ...
    }
    finally {
        System.out.println("Method methodOne() took: " + (System.currentTimeMillis()-start) + " ms");
    }
}
.
.
.
```

Later, we may wish to do the same for some other methods: "methodTwo()", "methodThree()" and so

on. This means we need to duplicate the code across many methods, which means lots of code changes. Also, the code we want to add is not really part of the business logic. It is more of a common utility code. If we can abstract this code into a separate method and invoke it automatically before any business method, that would make it very powerful and flexible. This is exactly what Aspect Oriented Programming at Method level does. It is the common code that cuts across the application and is invoked before any of the methods in the application.

Interceptors can be applied to either Session Beans or Message Driven Beans. An interceptor method can either be defined in the same class as the bean implementation class or it can be defined separately in another class that is not a bean implementation.

We will now look at an example that will use an interceptor using EJB3 annotations. In this hypothetical example, we want to track the average time it takes to generate an arbitrary token.

The following shows the code for the Remote interface called "**Token**":

```
/*
 * Name: Bhaskar S
 *
 * Date: 12/14/2008
 */

package token.remote;

import javax.ejb.Remote;

@Remote
public interface Token {
	public int getToken(String name) throws Exception;
}
```

The following shows the **TokenBean** stateless session bean implementation:

```
/*
 * Name: Bhaskar S
 *
 * Date: 12/14/2008
 */

package token.ejb;

import token.remote.Token;

import javax.ejb.Stateless;
import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;

import java.security.MessageDigest;

@Stateless(name="Token")
public class TokenBean implements Token {
	@Override
	public int getToken(String name)
		throws Exception {
		byte[] dg = null;

		MessageDigest md5 = MessageDigest.getInstance("MD5");

		StringBuffer sb = new StringBuffer();
		sb.append(name);
```

```
                sb.append(":");
                sb.append(System.currentTimeMillis());

                md5.update(sb.toString().getBytes());

                dg = md5.digest(name.getBytes());

                sb.append(":");
                sb.append(new String(dg));

                md5.update(sb.toString().getBytes());

                dg = md5.digest(name.getBytes());

                return Math.abs(new String(dg).hashCode());
        }

        @AroundInvoke
        public Object TimeMonitor(InvocationContext ctx)
                throws Exception {
            long stm = System.currentTimeMillis();
            try {
                return ctx.proceed();
            }
            finally {
                System.out.println("TimeMonitor Intercepted method " + ctx.getMethod().getName()
                + " took " + (System.currentTimeMillis()-stm) + " ms");
            }
        }
}
```

The @AroundInvoke annotation configures the method "**TimeMonitor**" as the interceptor method. In this example, we have defined the interceptor in the bean implementation class itself. The interceptor method has the following method signature:

```
public Object <method-name>(InvocationContext ctx) throws Exception;
```

where <method-name> is the name of the user-defined interceptor method and the argument is of type InvocationContext, which provides the context of the method to be invoked such as the method name, its parameters, etc.

The following shows a standalone client that is accessing the Token stateless session bean through the remote interface "**Token**":

```
/*
 * Name: Bhaskar S
 *
 * Date: 12/14/2008
 */

package token.client;

import token.remote.Token;

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

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

```
                try {
                        Context ctx = new InitialContext();

                        Token clnt = (Token) ctx.lookup("Token/remote");

                        System.out.println("Token: " + clnt.getToken(args[0]));
                }
                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 "**TokenClient.sh**" as illustrated below:

```
$ TokenClient.sh Bhaskar

Token: 1808784585

$ TokenClient.sh Swaminathan

Token: 181432309
```

From the JBoss Application Server log (server/default/log/server.log), we can see the following lines:

```
21:45:45,615 INFO  [STDOUT] TimeMonitor Intercepted method getToken took 1 ms
21:46:22,637 INFO  [STDOUT] TimeMonitor Intercepted method getToken took 0 ms
```

We will now look at another example that will use interceptors using XML deployment descriptors. Also, we will define our interceptor methods in external classes (not in the bean implementation). In this trivial calculator example, we want to log all the method invocations as well as the time taken by each of the invoked method of the bean.

The following shows the code for the method call logger interceptor called **AuditLogger**:

```
/*
 * Name: Bhaskar S
 *
 * Date: 12/14/2008
 */

package intercept.ejb;

import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;

public class AuditLogger {
        @AroundInvoke
        public Object auditLog(InvocationContext ctx)
                throws Exception {
            try {
                return ctx.proceed();
            }
            finally {
                StringBuffer sb = new StringBuffer();
                sb.append("AuditLogger Intercepted method: ");
```

```
                sb.append(ctx.getMethod().getName());
                sb.append("\n");
                Object[] args = ctx.getParameters();
                if (args.length > 0) {
                        sb.append("\twith parameters:");
                        for (int i = 0; i < args.length; i++) {
                                sb.append("\n\t\t");
                                sb.append(args[i].toString());
                        }
                        sb.append("\n");
                }
                System.out.println(sb.toString());
            }
        }
}
```

Notice that the interceptor is defined in a separate class called **AuditLogger**. This interceptor will log the method name along with its parameters.

The following shows the code for the other interceptor called **TimeProfiler**:

```
/*
 * Name: Bhaskar S
 *
 * Date: 12/14/2008
 */

package intercept.ejb;

import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;

public class TimeProfiler {
        @AroundInvoke
        public Object TimeMonitor(InvocationContext ctx)
                throws Exception {
            long stm = System.currentTimeMillis();
            try {
                return ctx.proceed();
            }
            finally {
                System.out.println("TimeProfiler Intercepted method " + ctx.getMethod().getName() + "
took " + (System.currentTimeMillis()-stm) + " ms");
            }
        }
}
```

Notice that this interceptor is also defined in a separate class called **TimeProfiler**. This interceptor will log the time taken by a method.

The following shows the code for the Remote interface called "**Calculator**":

```
/*
 * Name: Bhaskar S
 *
 * Date: 12/14/2008
 */

package intercept.remote;

public interface Calculator {
        public int add(int a, int b);
```

```
	public int subtract(int a, int b);
	public int multiply(int a, int b);
}
```

The following shows the **CalculatorBean** stateless session bean implementation:

```
/*
 * Name: Bhaskar S
 *
 * Date: 12/14/2008
 */

package intercept.ejb;

import intercept.remote.Calculator;

public class CalculatorBean implements Calculator {
	@Override
	public int add(int a, int b) {
		return a+b;
	}

	@Override
	public int subtract(int a, int b) {
		return a-b;
	}

	@Override
	public int multiply(int a, int b) {
		return a*b;
	}
}
```

Notice that neither the interface nor the bean implementation is using any annotations.

The following shows the XML deployment descriptor file that configures the above indicated regular Java classes as a stateless session bean and also configures the interceptors:

```
<?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>Calculator</ejb-name>
	     <remote>intercept.remote.Calculator</remote>
	     <ejb-class>intercept.ejb.CalculatorBean</ejb-class>
	     <session-type>Stateless</session-type>
	     <transaction-type>Container</transaction-type>
	 </session>
	</enterprise-beans>

	<assembly-descriptor>
	   <interceptor-binding>
	       <ejb-name>Calculator</ejb-name>
	       <interceptor-class>intercept.ejb.AuditLogger</interceptor-class>
	   </interceptor-binding>
	    <interceptor-binding>
	       <ejb-name>Calculator</ejb-name>
	       <interceptor-class>intercept.ejb.TimeProfiler</interceptor-class>
	    </interceptor-binding>
```

```
	</assembly-descriptor>
</ejb-jar>
```

From the above configuration file, it is clear that the EJB is named "**Calculator**" and it is a stateless session bean. In additional, the deployment descriptor indicates that the two interceptor classes: **AuditLogger** and **TimeProfiler** will be applied to the bean named "**Calculator**".

The following shows a standalone client that is accessing the "**Calculator**" stateless session bean through the remote interface:

```
/*
 * Name: Bhaskar S
 *
 * Date: 12/14/2008
 */

package intercept.client;

import intercept.remote.Calculator;

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

public class CalculatorClient {
	public static void main(String[] args) {
		try {
			Context ctx = new InitialContext();

			Calculator calc = (Calculator) ctx.lookup("Calculator/remote");

			System.out.println("(10 + 5) = " + calc.add(10, 5));
			System.out.println("(10 - 5) = " + calc.subtract(10, 5));
			System.out.println("(10 x 5) = " + calc.multiply(10, 5));
		}
		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 "**CalculatorClient.sh**" as illustrated below:

```
$ CalculatorClient.sh

(10 + 5) = 15
(10 - 5) = 5
(10 x 5) = 50
```

From the JBoss Application Server log (server/default/log/server.log), we can see the following lines:

```
14:59:50,092 INFO  [STDOUT] TimeProfiler Intercepted method add took 0 ms
14:59:50,092 INFO  [STDOUT] AuditLogger Intercepted method: add
	with parameters:
		10
```

```
              5
14:59:50,099 INFO  [STDOUT] TimeProfiler Intercepted method subtract took 0 ms
14:59:50,099 INFO  [STDOUT] AuditLogger Intercepted method: subtract
       with parameters:
              10
              5
14:59:50,102 INFO  [STDOUT] TimeProfiler Intercepted method multiply took 0 ms
14:59:50,103 INFO  [STDOUT] AuditLogger Intercepted method: multiply
       with parameters:
              10
              5
```

This concludes our journey into EJB Timers and Interceptors.

We will explore EJB Entity Beans Using Java Persistence in Part-6 of this series.