

CS420 – Software Engineering in Practice
Lab 2 - Creating and Deploying Enterprise Java Beans into JBOSS
Due: End of class
(40 pts)

Team up with one other classmate to complete this assignment.

I. Review and Background:

Building an EJB requires assembling the correct component parts. These include interfaces and deployment descriptors.

EJB Interfaces

- All bean classes must implement `javax.ejb.EnterpriseBean`, or one of its subinterfaces, at some level.
 - `javax.ejb.EnterpriseBean` extends `java.io.Serializable`.
- Session beans implement `javax.ejb.SessionBean`.
 - `javax.ejb.SessionBean` extends `javax.ejb.EnterpriseBean`.
 - It defines methods: `ejbActivate`, `ejbPassivate`, `ejbRemove`, and `setSessionContext`.
- Entity Beans implement `javax.ejb.EntityBean`.
 - `javax.ejb.EntityBean` extends `javax.ejb.EnterpriseBean`.
 - It defines methods: `ejbActivate`, `ejbPassivate`, `ejbRemove`, `setEntityContext`, `ejbLoad`, `ejbStore`, and `unsetEntityContext`.
- Message Driven Beans implement `javax.ejb.MessageDrivenBean`.
 - `javax.ejb.MessageDrivenBean` extends `javax.ejb.EnterpriseBean`.
 - It defines methods: `ejbRemove` and `setMessageDrivenContext`.

What Constitutes an EJB?

- **EJB Object**
A client never invokes methods directly on an actual bean instance. All invocations go through the EJB object, which is a tool-generated class. Message requests are intercepted by the EJB object and then delegated to the actual bean instance. That is why it is called a request interceptor.
- **Remote Interface**
A bean class exposes business methods that are cloned by the EJB object. But how do the tools that generate the EJB object know which methods to clone? The answer is the remote interface. This interface duplicates all the methods that the corresponding bean class exposes (remember that they must comply with the EJB specification). Remote clients then access the bean via this remote interface. All remote interfaces must extend `javax.ejb.EJBObject`.
- **Home Object**
How do clients acquire references to EJB objects? The client asks for the EJB object from the EJB object factory. This factory is responsible for instantiating and destroying the EJB objects. This factory is called the Home object.
- **Home Interface**
How does a home object know how you would like to initialize your EJB object?

This information is provided to the container through the home interface. The home interface contains the methods for creating, destroying, and finding EJB objects. All home interfaces are derived from the `javax.ejb.EJBHome` interface and all home objects implement the home interface. This interface extends `java.rmi.Remote`.

- **Local Interfaces**
Creating beans through the home interface and then calling the beans through the remote interface is very slow. Local interfaces speed up beans' calls. Use local objects (implemented from a local interface rather than from a remote interface) instead of EJB objects whenever possible. Also, use local home objects (implemented from a local home interface rather than from a home interface) instead of home objects. Local interfaces extend `javax.ejb.EJBLocalObject` and local home interfaces extend `javax.ejb.EJBLocalHome`.
- **Deployment Descriptors**
You state your deployment configuration in the deployment descriptor file named `ejb-jar.xml`. Different application servers may have their own additional proprietary deployment descriptor files. These XML files can either be edited by hand or with a tool provided by the container vendor. You can specify:
 - Bean management and lifecycle requirements.
 - Persistence requirements for entity beans.
 - Transaction requirements.
 - Security requirements, including access control entries.
- **EJB-JAR file**
Once you have finished compiling your classes and creating all of the necessary configuration files, you can place all of these files into an EJB-JAR file for deployment. EJB-JAR files are zip files that contain Java classes and all the other files needed for deployment. You can generate these files by hand or with tools, such as Apache Ant.

Next we will create an EJB, deploy it, and then run it under jboss.

II. Lab Instructions:

Before we go further, the following code has been tested on both jboss 3.2.5 and 4.0.1. To follow along, download and install one of these jboss versions from <http://www.jboss.org/downloads/index>. You can get jboss version 4.0.1 off of the class website.

The Directory Structure

Create the following directory structure under some logical parent directory, such as `c:\ejb_example`. This directory structure is for development purposes only; the deployment files will be zipped into a jar file in a moment and deployed to jboss.

```

src/
|__client/
|  |__com/
|  |  |__examples/
|  |  |__client java files
|__server/
|  |__com/
|  |  |__examples/
|  |  |__server (bean) java files
|__shared
|  |__com/
|  |  |__examples/
|  |  |__remote and home java files
|
assemble/
|__client and server jars
|
target/
|__client/
|  |__com/
|  |  |__examples/
|  |  |__client, remote and home java classes
|  |  |__jndi.properties
|__server/
|  |__com/
|  |  |__examples/
|  |  |__server (bean), remote and home java classes
|  |__META-INF/
|  |  |__ejb-jar.xml

```

The Bean Class

Create the bean class as `src\server\com\examples\HelloBean.java`. This class is a stateless session bean and contains our business logic. In this case it prints out the exciting and always useful "Hello! World".

```

package com.examples;

import javax.ejb.*;

public class HelloBean implements SessionBean
{
    public void ejbCreate() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbRemove() {}
    public void setSessionContext(SessionContext sc){}
}

```

```
public String sayHello ()
{
    System.out.println ("Someone called sayHello()");
    return "Hello! World";
}
}
```

The Remote Interface

Create the remote interface as src\shared\com\examples\Hello.java. This is the interface that remote clients talk to instead of talking directly to the bean class, HelloBean. Notice that its only method is sayHello(), which is the business method in HelloBean.

```
package com.examples;

import javax.ejb.*;
import java.rmi.*;

public interface Hello extends EJBObject
{
    public String sayHello() throws RemoteException;
}
```

The Home Interface

Create the home interface as src\shared\com\examples\HelloHome.java. This interface will be used to create an instance of the Hello interface when we want to execute the sayHello() business logic.

```
package com.examples;

import javax.ejb.*;
import java.rmi.*;

public interface HelloHome extends EJBHome
{
    public Hello create() throws CreateException, RemoteException;
}
```

The Client

Create the client as src\client\com\examples\HelloClient.java. This is the "remote" client that will use our session bean. First, it uses the InitialContext to get a handle to HelloHome. It then uses the HelloHome interface to create hello. hello is a remote interface representing our bean class. After hello is used to execute the business logic (sayHello), the bean is released via the remove method.

```
package com.examples;

import javax.ejb.*;
import java.rmi.*;
import javax.rmi.*;
import javax.naming.*;

public class HelloClient
{
    public static void main (String[] args) throws Exception
    {
        Context c = new InitialContext();
        Object o = c.lookup ("Hello");

        HelloHome home =
            (HelloHome)PortableRemoteObject.narrow (o,HelloHome.class);
        Hello hello = home.create();
        System.out.println (hello.sayHello());
        hello.remove();
    }
}
```

ejb-jar.xml

Create the ejb-jar.xml file in target\server\META-INF. This deployment descriptor contains information that the jboss EJB container needs in order to deploy and run our EJB.

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
    <enterprise-beans>
        <session>
            <ejb-name>Hello</ejb-name>
            <home>com.examples.HelloHome</home>
            <remote>com.examples.Hello</remote>
            <ejb-class>com.examples.HelloBean</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>
        </session>
    </enterprise-beans>
```

```
</ejb-jar>
```

jndi.properties

Create the `jndi.properties` file in `target\client`. This file defines properties that are required in order to use JNDI to find EJBs on the network.

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=jnp://localhost:1099
```

Now we have all of the required files created in our directory structure. In the next and final step we will compile our code, deploy it, and then run in in jboss.

Compile the Source Code

Now we have all of our files in the correct directories. The next step is to compile the source code into class files. The following commands should be executed from the top of the directory structure created earlier (`c:\ejb_example` on my computer). Also, you'll need to ensure your `javac` command is working and that you have the proper development kits installed. (These commands are for Microsoft Windows. If running this example on another operating system, just edit these commands as required.) The `LIBDIR` variable should point to the directory that contains `jboss-j2ee.jar`.

(The line breaks are for formatting only. Execute these on one line.)

```
javac -classpath %LIBDIR%\jboss-j2ee.jar -d
    target\client src\client\com\examples\*.java
    src\shared\com\examples\*.java
```

```
javac -classpath %LIBDIR%\jboss-j2ee.jar -d
    target\server src\server\com\examples\*.java
    src\shared\com\examples\*.java
```

Create the Jar Files

Remember that EJBs execute in an EJB container on the server. In our case this will be the jboss application server. We need to package the server code into a jar file

named `helloserver.jar`. This jar will be copied into the proper jboss deployment directory in the next step. Likewise, we need to package the client code, the code that will call and use the EJB, into a jar file named `helloclient.jar`. The client code can be run from any computer on the network, but in our case we'll just run jboss in one DOS window and the client in another DOS window on the same computer.

Execute the following command from your `c:\ejb_example\target\client` directory. (Don't forget the trailing "." at the end.)

```
jar cvf ../../assemble/helloclient.jar .
```

Execute the next command from your `c:\ejb_example\target\server` directory. (Again, there is a trailing ".")

```
jar cvf ../../assemble/helloserver.jar .
```

After executing these jar commands you should have a `helloclient.jar` and a `helloserver.jar` in the `assemble` directory.

Deploy to jboss

1. Create a `JBOSS_HOME` environment variable and set it to the jboss installation directory (for instance, `c:\jboss-3.2.5`).
2. Create a `LIBDIR` environment variable and set it to the client directory under `JBOSS_HOME` (for instance, `%JBOSS_HOME%\client`).
3. Place the `helloserver.jar` in the `%JBOSS_HOME%\server\default\deploy` directory.
4. From the command prompt:
5. `%JBOSS_HOME%\bin\run.bat`

When jboss starts up, you should see logging statements indicating that the Hello EJB has been deployed. Here is what I have in my command window:

```
08:56:07,656 INFO [EjbModule] Deploying Hello
08:56:08,424 INFO [EJBDeployer] Deployed: file:c:\jboss-
4.0.1sp1\server\default\deploy\helloserver.jar
```

Run the Client

To run the client, open a new DOS window (you should have jboss already running in a separate window).

1. Create a `JBOSS_HOME` environment variable and set it to the jboss installation directory (for instance, `c:\jboss-3.2.5`).
2. Create a `LIBDIR` environment variable and set it to the client directory under `JBOSS_HOME` (for instance, `%JBOSS_HOME%\client`).

3. Change directory to the assemble directory that contains helloclient.jar.
4. From the command prompt (all as one line):
5. `java -classpath helloclient.jar;%LIBDIR%\jnp-client.jar;%LIBDIR%\jboss-common-client.jar;%LIBDIR%\jboss-j2ee.jar;%LIBDIR%\jboss-net-client.jar;%LIBDIR%\jbossall-client.jar;;%LIBDIR%\jnet.jar com.examples.HelloClient`

In your client DOS window, you should see "Hello! World" print out. In the server DOS window, you should see "Someone called sayHello()" print out. The client just executed our sayHello business method from within the jboss application