Savaş Ali TOKMEN
http://ali.tokmen.com/

Web services and EJB3s in JOnAS 4

IMPORTANT: There have been major changes in the Java EE world since this article has been written. In particular, JOnAS 5 is out and integrates EJB3s as well as JAX-WS Web Services. **These render the deployment of EJB3s and deployment of Web Services much easier than in this article.**

Summary: In this article I'll be describing how to implement a web service and EJB3s in JOnAS 4. The two parts are independent; therefore you can use this article whether you're trying to do a web service, some EJB3s, or both. I'm writing this because I didn't find anywhere else this information is present and thought people would appreciate having a "simple" documentation explaining some things (even if explanations are not complete) + a sample project that can be used as a start point.

Me, you and the aim

I, Savaş Ali Tokmen, am a student at the UFR IMA, in Grenoble (France). As part of my Masters degree, we do a lot of "practical" work in multiple areas (and I would again like to thank my professors for that!) and one of these is to implement a distributed measurement environment. The point that's important for us (and this article) is that some data are sent to a Java EE server using a web service and then saved into a database using EJB3s. I won't be presenting you the web service as implemented for that project, otherwise next years' students would shamelessly steal the work and don't learn anything :)

Languages I've worked with include a lot, but the ones I really work with would be C++, PHP and DHTML. I've been using the Visual Studio environment for years (and like it a lot), and for the last two years I've also been working with Eclipse and the Java DK. I'm therefore not the expert on the domain, and on some approaches you'll perhaps say "hey, that's a C++ style!" (for example, for the package names I choose).

You are a person who's been working with Java for some time (a year maybe), and have some knowledge on Java EE. In this article, I won't tell you to use .net or JBoss or some other technology. As part of my course, I've worked on JOnAS 4 (namely JOnAS 4.7.4 and JOnAS 4.8.4) and all I'll be mentioning in this article has only been tested on that.

The aim has two parts: first, to create a web service in JOnAS. Second, to link that web service to EJBs. Ready, set; and let's go!

Web service

What is a web service?

Well, you've probably used Java RMI or CORBA at least once right? The aim in such technologies is to call a function not on the local machine but on a distant one. You could for example have a "white pages" server which has a "lookup" function (I'm taking this example because I've implemented it before. Well, with CORBA or RMI you can call that function from your machine, and it would execute on the server and the result would be brought back to you.

In a web service, instead of using RMI or CORBA, what we use is the HTTP protocol (yes, the exact same one as the one used for web pages!). In order to call a function, you will therefore use URLs. When you need to pass arguments to that function, you'll be using XML.

With Java EE, what's magic is that **you don't need to know at all how HTTP or XML works in order to understand how to create a web service**!

Creating the web service interface

In a RMI or CORBA application, what you first need to do is to decide how your functions are called (by this I mean the name of the functions and of the arguments, nothing more).

JOnAS 4 uses a technology called "JAX-RPC" (Java API for XML-Based Remote Procedure Call) in order to implement web services. That technology is quite old (2002) and requires the programmer to take a certain number of manual steps (on the other hand, you can write ANT tasks which would do all these steps in one step). JAX-WS is easier, but is in Java EE version 5. Therefore, you'll have to use JAX-RPC if you want to use JOnAS 4 without doing heavy modifications on it.

The good part with JAX-RPC is that it uses the exact same syntax as RMI. I'll now describe a web service which receives sales data for a given item:

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface SalesArchivingService extends Remote {
    public void sendSalesData( SalesItem item, SalesData[] datum )
                throws RemoteException;
}
```

In that example, **SalesItem** is a simple class which has one String (the name of the product) and **SalesData** contains the Date of the sale and the price at which the item has been sold (we suppose that the sell price changes at every sale -that generally happens in small shops or in a stock exchange, for instance). Of course, typing all this is useless and annoying: that's why all source codes are available on http://scholar.alishomepage.com/Master/WS_JOnAS4/

Of course, a web service doesn't use Java interfaces! Instead, it uses a language called WSDL (Web Service Definition Language). As with the SOAP requests, a WSDL is also written is XML.

And, of course, you don't want to write any piece of code to decode XML (mostly that I had told you wouldn't). In order to generate the WSDL file based on your Java interface, you can use **java2wsdl**. If you love command lines, you can type it by hand at each time. I hate typing and use ANT instead. Here's part of my ANT build file:

```
<taskdef name="axis-wsdl2java"
    classname="org.apache.axis.tools.ant.wsdl.Wsdl2javaAntTask">
    <classpath refid="axis.classpath" />
</taskdef>
<target name="wsdl" depends="compile">
    <axis-java2wsdl classname="${service_name}.${java2wsdl.class_name}"
                    location="http://dummy-url"
                    serviceportname="${service_name}Port"
                    namespace="urn:${service_name}"
                    typeMappingVersion="1.3"
                    output="${xml.dir}/wsdl/${service_name}.wsdl">
        <classpath refid="base.classpath" />
    </axis-java2wsdl>
</target>
```

Again, don't try to copy-paste that code. It's all on my web site. Download it!

Java2WSDL has some important arguments. Here are some details about them:

- **location** is the web server at which the web service will be running (like http://192.168.4.5). Setting that value is totally useless as JOnAS will automatically generate the appropriate address by itself.
- **serviceportname** is the part of the URL that'll come after the web server's address. I'll detail it a bit more later on.
- **namespace** is the WSDL package name you'll be using. Using the same name as you Java classname would be wise (results in less confusion).

Deploying the web service

For now, the only thing we have in hand is a Java interface and a semi-complete WSDL generated using it. We now need to have two things: something that implements the web service and another thing that would actually listen for SOAP calls (which are some HTTP requests, exactly as loading web pages, remember). Implementing the web service "for real" will be hard for now, we don't have our EJBs. But there's a simple implementation that's perfectly valid:

```
public class SalesArchivingServiceImpl implements SalesArchivingService {

    public void sendSalesData( SalesItem item, SalesData[] datum )
                throws RemoteException {
        System.out.println("Web service received something!");
    }

}
```

Very advanced implementation, isn't it? :) Now, we need to tell JOnAS that this is the implementation of the web service. To do that, we need to create 3 more XML files:

- **mapping.xml** file, where we tell how to map Java packages to WSDL namespaces. Since you've done like I did and used the same name for both, that's not too hard:

```
<?xml version="1.0"?>
<java-wsdl-mapping xmlns="http://java.sun.com/xml/ns/j2ee"
                   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                       http://www.ibm.com/webservices/xsd/j2ee_jaxrpc_mapping_1_1.xsd"
                   version="1.1">
    <package-mapping>
        <package-type>SalesArchivingService</package-type>
        <namespaceURI>urn:SalesArchivingService</namespaceURI>
    </package-mapping>
</java-wsdl-mapping>
```

  In my files, that file is called **SalesMapping.xml**.

- **web.xml** file, where we tell how to map the **SalesArchivingServiceImpl** class to a URL:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                             http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
         version="2.4">

  <display-name>Sales Archiving Web Service</display-name>
  <servlet>
    <display-name>Sales Archiving Web Service</display-name>
    <servlet-name>SalesArchivingServlet</servlet-name>
    <servlet-class>SalesArchivingService.SalesArchivingServiceImpl</servlet-class>
  </servlet>
</web-app>
```

  JOnAS will automatically implement that URL as a servlet. That servlet will fetch the SOAP request, decode the XML in it and call your implementing class with normal Java objects -which is why you don't need to know how HTTP, SOAP or XML parsing works in order to do a web service!

- **webservices.xml** file, where you tell how the WSDL, the web service and all those things actually get together. I won't paste that one, just look in my files. You'll probably feel annoyed when building this file, as it it just repetition of the names you've given in the **web.xml** and **SalesMapping.xml** files.

As you know, in the Java EE world, deploying a web application is deploying a WAR file which is a ZIP file with all your Java classes, XML descriptors and resources. In order to have the application ready, we therefore need to create the WAR.

To create a WAR, we use the **war** ANT task:

```
<target name="wars" depends="compile, wsdl">
    <war warfile="${temp.webapps.dir}/${service_name}.war" webxml="${xml.dir}/web.xml">
        <classes dir="${classes.dir}" />
        <webinf dir="${xml.dir}">
            <include name="wsdl/*.wsdl" />
            <include name="*.xml" />
            <exclude name="web.xml" />
        </webinf>
    </war>
</target>
```

Once the WAR is generated, you'll need to create the associated web service and plug it into the WAR. This is done using the **wsgen** tool:

```
<target name="wsgen" depends="wars">
    <wsgen srcdir="${temp.dir}" destdir="${dist.dir}" verbose="false" debug="false">
        <include name="webapps/${service_name}.war" />
    </wsgen>
</target>
```

After those steps, our web service is ready to be deployed! Just go to the JOnAS administration panel, upload your WAR and deploy. If you go to the "web services" page in the JOnAS administration panel, you should see that your web service is operational.

Let's now make a first application to call it.

Using the web service

The JOnAS administration panel should tell you that the application has been deployed on a URL such as http://localhost:9000/SalesArchivingService/SalesArchivingWebService/SalesArchivingServicePort?JWSDL . That's the sum of all I've written in those nasty XML files. In order to get the WSDL file, you can put a **?JWSDL** at the end of that URL and JOnAS will generate and send you the WSDL.

You can look at the WSDL, it's actually easy to read if you have an XML reader (Internet Explorer and Firefox both read XML perfectly). On the other hand, writing the appropriate Java classes based on the WSDL would be such a pain! Luckily, **wsdl2java** does it for us:

```
<taskdef name="axis-wsdl2java"
         classname="org.apache.axis.tools.ant.wsdl.Wsdl2javaAntTask">
    <classpath refid="axis.classpath" />
</taskdef>

<target name="wsdl2java">
    <input message="Enter name or URL of the WSDL file to extract"
           addproperty="wsdlFN"
           defaultvalue="http://localhost:9000/SalesArchivingService/
                         SalesArchivingWebService/SalesArchivingServicePort?JWSDL"
     />
    <mkdir dir="wsdl2java" />
    <axis-wsdl2java output="wsdl2java" url="${wsdlFN}" />
</target>
```

The **wsdl2java** task I've just pasted here will extract the WSDL's contents into the **wsdl2java** directory. It will actually generate a bit more than that:

- We have the **SalesItem** and **SalesData** classes as well as the **SalesArchivingService** interface. That's expected, since those classes were what we had used to generate the WSDL. There's one difference you can note: the data's **date** field's type has changed from **Date** to **Calendar**. That's because AXIS maps the WSDL Date definition to a Java Calendar.
- We also have the **Service**, **ServiceLocator** and **PortSoapBindingStub** classes. Those classes, which use the AXIS API in our case, will actually be doing the localize the web service + convert the Java arguments to XML + fetch the result part. This is why you don't need to know anything about HTTP, XML encoding / decoding or about SOAP in order to do a web service with Java EE!

Since the ANT task has done the hard work for us, writing a client is really easy:

```
public class SalesArchivingClient {
    public static void main(String[] args) {
        try {
            // Find the web service
            SalesArchivingServiceInterface service = new
                SalesArchivingServiceInterfaceServiceLocator().
                getSalesArchivingServicePort();

            // Create data to send
            SalesItem item = new SalesItem("test");
            SalesData[] datum = new SalesData[10];
            for(int i=0 ; i<10 ; i++) {
                datum[i] = new SalesData(new GregorianCalendar(), i+1);
            }

            // Send data
            service.sendSalesData(item,datum);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

As a result, you just locate the web service and then make your Java calls on the web service, exactly as if the object was local. Simple.

Conclusion

Well, once you know how to do it, creating a web service + the client for it is very simple with JAX-RPC and JOnAS: you need to write **no more than 350 lines** of Java, XML and ANT in total. In our case, the client will "only" need to have the AXIS API present, which requires about 1.5 MB of storage space.

With JAX-WS, things will get easier: most of those extra XML files will be stripped off! You'll therefore be able of putting all the mambo-jumbo in your Java code. I guess this example will fit in 200 lines of code once JAX-WS is in JOnAS!

You'll find the source codes of this part in the **SalesWebService.zip** file, that you can download from my web site: http://scholar.alishomepage.com/Master/WS_JOnAS4/ . Please feel free to use and distribute this code as you wish.

EJB3

What is an EJB3?

EJB is the short for "Enterprise Java Bean". It's nothing more than a Java class that can represent different parts of the application: it can be a tuple in a database, it can be a transaction manager, etc. In previous versions of EJB, what we used to do was to write the EJB classes, then write the corresponding XML files that tells how the EJBs will actually be deployed. With EJB3, XML files became Java annotations; which makes programming and coding easier.

In this application, we'll be defining two types beans:

- Some entity beans, which represent our items and sales data that are stored in the database. Entity beans can be seen as tuples in the database.
- The container bean, which creates the entity beans and saves them into the database. It The container bean can be seen as a transaction manager.

In order to have an EJB3 bean that will save incoming data to the database, we first create its interface:

```
public interface SalesArchivingBeanRemote {
    public void saveSalesData(SalesItem item, SalesData[] datum) throws Exception;
}
```

Now that we have the interface of the container bean, the web service can call it:

```
public class SalesArchivingServiceImpl implements SalesArchivingServiceInterface {
    public void sendSalesData(SalesItem item, SalesData[] datum)
           throws RemoteException {
        try {
            Context ctx = new InitialContext();
            SalesArchivingBeanRemote containerBean = (SalesArchivingBeanRemote)
                ctx.lookup("SalesArchivingService.SalesArchivingBean_" +
                           "SalesArchivingService.SalesArchivingBeanRemote@Remote");
            containerBean.saveSalesData(item, datum);
        } catch(Exception e) {
            throw new RemoteException("Error saving sales data: "+e);
        }
    }
}
```

The implementation of the container bean is as follows:

```
@Stateless
@Remote(SalesArchivingBeanRemote.class)
public class SalesArchivingBean implements SalesArchivingBeanRemote {
    @PersistenceContext
    private EntityManager entityManager = null;

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void saveSalesData(SalesItem item, SalesData[] datum) throws Exception {
        // Try to find the item, if not present create it
        SalesItemBean itemBean = null;
        {
            Query itemQuery = entityManager.createQuery("select object(b) from " +
                                                        "SalesItemBean b where " +
                                                        "b.name = :name");
            try {
                itemQuery.setParameter("name", item.name);
                itemBean = (SalesItemBean) itemQuery.getSingleResult();
            } catch (Exception e) {}

            if(itemBean == null) {
                itemBean = new SalesItemBean();
                itemBean.setName(item.name);
                entityManager.persist(itemBean);
                Logger.global.info("Will insert sales item bean: " + itemBean);
            }
        }

        // Add all sales data. Note that we don't need to add each data to the
        // item's "sales" list, as the the entity manager will fetch those and
        // automagically create the associated list as needed
        if(datum != null) {
            for( int i=0 ; i<datum.length ; i++ ) {
                SalesDataBean salesDataBean = new SalesDataBean();
                salesDataBean.setDate(datum[i].date);
                salesDataBean.setPrice(datum[i].price);
                salesDataBean.setItem(itemBean);
                entityManager.persist(salesDataBean);
                Logger.global.info("    Will insert sales data bean: " + salesDataBean);
            }
        }
    }
}
```

Like with JAX-RPC, you don't need to know how the underlying database works or how the listener will find the transaction manager. You just write your Java code, and it will automagically be linked together with your database.

Installing support for EJB3

Good news is, some people have done a package called EasyBeans which integrated EJB3 support into JOnAS 4. It can be found on http://www.easybeans.org/ and it installs very simply on JOnAS 4. Remember, that wasn't the case for JAX-WS...

On the other hand, deploying your EJB3s on JOnAS is harder.

Deploying an EJB3 application

In order to deploy our web service, what we needed to do is to put all needed files in a WAR, upload it to JOnAS and deploy it using the JOnAS administration console.

Unfortunately, EJB3s are not deployed with your WAR. As a result, if you try to deploy the **SalesArchivingBean** inside a WAR, the **@PersistenceContext** annotation will be ignored as a result the **EntityManager** will always be **null**.

What you need to do instead is to deploy your EJB3 classes together with a file called **persistence.xml** (which tells how to connect to the database) as a JAR. Here's how it goes:

- Create a directory in **JONAS_ROOT/ejb3s** with the name you want. I would recommend you to put the same name as your WAR, but with a .jar at the end.
- Put your .class files in that folder.
- Create a directory called **META-INF** in your JAR's folder and put the **persistence.xml** file in it. My persistence.xml looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
         http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
         version="1.0">

    <persistence-unit name="entity">
        <provider></provider>
        <jta-data-source>jdbc_1</jta-data-source>
        <properties>
            <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
        </properties>
    </persistence-unit>
</persistence>
```

Please note that the persistence.xml I've given uses as hibernation type "**create-drop**", which means that the database will be emptied everytime JOnAS is restarted! That's OK (even good) for testing, but remember to change this value once you finish testing.

In order to activate the JAR (whether I put it for the first time or did some small changes on it and redeployed), I needed to restart my JOnAS server. I guess there's a better (=faster) method for that as well, but remember to restart if JOnAS doesn't seem to have activated the beans you've modified.

Of course, all those annoying "rmdir-mkdir-copy-paste" steps can become easy to use ANT tasks:

```
<target name="install.persistence">
    <mkdir dir="${ejbjars.dir}/${service_name}.jar/META-INF/" />
    <copy todir="${ejbjars.dir}/${service_name}.jar/META-INF">
        <fileset dir="${xml.dir}">
            <include name="persistence.xml" />
        </fileset>
    </copy>
</target>

<target name="install.jar" depends="install.persistence">
    <copy todir="${ejbjars.dir}/${service_name}.jar">
        <fileset dir="${classes.dir}">
            <include name="**/*.class" />
        </fileset>
    </copy>
</target>
```

Conclusion

Once again, once you know how to do it, deploying an EJB3 application in JOnAS is as easy as a pie. The real "trick" you need to know is that you need to deploy the WAR for the web part of the application, and the JAR for the EJB3 part of your application. If you do everything manually that's extremely annoying, but if you do like I do and create ANT tasks for it the only thing you need to run is "**ant install**".

You'll find the source codes of this part in the **SalesWebService.zip** file, that you can download from my web site: http://scholar.alishomepage.com/Master/WS_JOnAS4/ . Please feel free to use and distribute this code as you wish.

Final part: put all these together

Now that you know how to deploy and active all those service, there's one last step that's annoying: actually writing all those classes and implementations.

Unfortunately, you cannot use Java's marvellous “extends” or “implements” features in EJB3. On the other hand, the classes the EJB3 will be working with probably are the same as the ones the web service will be working with (same attributes, same names; the only difference is the usage). You therefore have two choices:

1. Create an entity bean that simply delegates to the original class (the one used by the web service); as a result you won't need to bother on keeping those two classes up to date. That's a very bad idea since the data that will be saved on the database will be a serialized Java object: you won't be able of searching on it, and you'll have real trouble reading it from anything that's not Java.
2. Create a bean that does exactly the same thing (=copy-paste the class' code and put all needed EJB annotations on the copy). In this case, you'll have to manually keep the two classes up to date, on the other hand the bean will be a standard entity bean that behaves nicely with the database and with other languages.

Here's how the second solution goes for the **SalesItem** class (which ends up having another class called **SalesItemBean**):

```
@Entity
public class SalesItemBean {
    @Id
    @Column(name="iid")
    @GeneratedValue
    private long id;

    private String name;

    @OneToMany(mappedBy="item")
    private Set<SalesDataBean> sales;

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Set<SalesDataBean> getSales() {
        return this.sales;
    }

    public void setSales(Set<SalesDataBean> sales) {
        this.sales = sales;
    }

    public String toString() {
        return ( this.name );
    }
}
```

If you're lucky, Eclipse can generate you the bean based on the initial class. On the other hand, keeping the two classes up to date can get annoying. There is no solution I know that can handle this, I just hope one day EJBs will support inheritance!

Once again, you'll find the source codes of this part in the **SalesWebService.zip** file, that you can download from my web site: http://scholar.alishomepage.com/Master/WS_JOnAS4/ . Please feel free to use and distribute this code as you wish.

You can also use that file as a start point for your future web service & EJB3 project.