Service-enable EJB SessionBeans with the IBM ETTK

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

1. About this tutorial 2
2. Step-by-step Hello World example 5
3. Returning more complex types 24
4. Summary & further resources 31

Section 1. About this tutorial

Purpose of this tutorial

This tutorial shows J2EE developers how to use the IBM ETTK (Emerging Technologies Toolkit) to make any Enterprise JavaBean (EJB) component into a Web service that will run on any application server.

Do you currently use EJB technology? Have you ever wondered how to expose an EJB component as a Web service no matter what your application server is? This tutorial shows how to use the ETTK to convert a local and remote EJB component into a Web service. I will use a subcomponent of the ETTK called Axis, a SOAP client implementation originally hosted by the Apache Software Foundation.

The ETTK implements the Web services architecture and provides a set of tools to create, locate and invoke Web services. It includes the following tools:

- UDDI4J API allows developers to perform save, delete, find, and get operations against a UDDI registry.
- UDDI4J-WSDL API allows developers to perform publish, unpublish, and find operations against a UDDI registry.
- WSDL4J allows developers to programmatically read and work with WSDL documents.
- WSDLdoc allows developers to automatically generate documentation from WSDL files similar to JavaDoc.
- WSIL4J allows developers to programmatically read and work with WSIL documents.
- Axis RC1 allows a developer to generate Web services WSDL definitions from Java code and generate Java proxy code from a WSDL definition. Axis is also the transport engine for Web services.
- Sample services including Accounting, Contract, Metering, Service Desk etc.
- Demos and tutorials to illustrate key Web service concepts.
- Privacy Policy Authorization Director (PAD) Web service allows privacy policy control access to personal information.
- And much more...

The primary focus of this tutorial is developing EJB-based Web services with Axis. Axis provides support for turning EJB components into Web services and is included with the ETTK.

What do I need to know for this tutorial?

This tutorial assumes you have a working knowledge of Java technology and EJB technology. In-depth knowledge of EJB components, Web services, and Ant are helpful but not required to understand the key concepts. Ant is used to build and deploy the example applications. Reference to introductory material on Ant, Java technology, J2EE, Web services, XML, and EJB components are provided in throughout the tutorial and at the references section at the end of this tutorial.

The source code in the tutorial has been tested with *Resin EE* application server.

The applications should be easy to port to other J2EE-compliant application servers like *IBM WebSphere* or *JBoss*. Please check back at my site for ports to other application servers (see Resources on page 31).

I typically get examples from people working with other application servers, and then I put them up on my site (see Resources on page 31).

I used the Eclipse framework to create the examples in this tutorial. The examples are easiest to run by downloading Eclipse 2.1 or higher and a J2EE application server plug-in for Eclipse. Eclipse has excellent support for Ant, which facilitates running the Axis and XDoclet Ant tasks right from the IDE environment.

If you are new to Ant, please read this sample chapter from *Mastering Tomcat on Developing Web Components with Ant* (written by yours truly). Just read the sections on Ant development for now.

Also note that the Ant scripts can use XDoclet. You will not need to use this functionality, but, in case you decide to, please refer to the developerWorks tutorial, "Enhance J2EE component reuse with XDoclets" (see Resources on page 31).

What this tutorial covers

This tutorial covers turning EJB components into Web services and has two step-by-step examples. The first example uses all primitive types with one simple EJB component. The second example uses a SessionBean that talks to several EJB components. All examples ship with a set of Ant build scripts so you can easily create your own custom solutions by reusing the sample build files.

About the author

Rick Hightower is a developer who enjoys working with Java technology, Ant, ETTK, and XDoclet. Rick is currently the CTO of *Trivera Technologies*, a global training,

mentoring, and consulting company focused on enterprise development.

If you like this tutorial, you might like Rick's book, *Java Tools for Extreme Programming*, which was the best-selling software development book on Amazon for three months in 2002 and covers applying Ant, JUnit, Cactus, and more to J2EE development. You might also like to read other IBM *developerWorks* tutorials that Rick helped produce and author (see Resources on page 31).

Rick also contributed two chapters to the book *Mastering Tomcat* on the subjects of Struts Tutorial and Tomcat development with Ant and XDoclet, as well as many other publications.

Rick is also speaking this year (2003) at JavaOne on EJB CMP/CMR and XDoclet and at TheServerSide.com Software Symposium on J2EE development with XDoclet. Rick has spoken at JDJEdge, WebServicesEdge, and the Complete Programmer Network software symposiums.

Tools you will need for this tutorial

You will need a current version of the JDK. All of the examples in this tutorial use J2SE SDK 1.4.1.

All of the examples use Ant build files to build and deploy the Web applications that contain the examples. This should be no surprise since XDoclet relies on Ant, and the only interface to XDoclet is through Ant. Ant can be found at the *Ant home page*. The examples use Ant 1.5.3.

Of course you will need the IBM ETTK. The ETTK can be found at the *ETTK site*. The examples in this tutorial use version ETTK 1.0, which includes Axis 1.1 release candidate 2.

It is recommended that you use an Interactive Development Environment (IDE), like Eclipse, since there are quite a few jar files to manage (this is quite an understatement). All the examples ship with the projects done in the freely-available Eclipse IDE and are compatible with Eclipse and WebSphere Studio Application Developer. As long as you configure your environment as suggested, you can use the Eclipse project files with little additional work. Eclipse or WebSphere Studio Application Developer is not required, but can be found at the *Eclipse Web Site* and *WebSphere Studio Application Developer trial* respectively. There is no requirement to use Eclipse, but the Eclipse project files are provided as a convenience to Eclipse-based users.

If you do not use an Eclipse-based system, please use some IDE. It saves you time in the long run.

Section 2. Step-by-step Hello World example

Steps

Prime your engines by creating a simple EJB component that uses Axis to expose itself as a Web service. To do this you will need to perform the following steps:

1. Download and install the ETTK
2. Create a simple EJB component
3. Create a Web application and configure its deployment descriptor (*web.xml*)
4. Create a Web service deployment descriptor for your EJB component
5. Use Axis Ant tasks to deploy the EJB-based Web service
6. Use Axis Ant task **wsdk2java** to generate client stubs
7. Write a Web service client for your EJB component
8. Work with Local EJB Services.

Download and install the ETTK

Read this complete page before starting the ETTK download.

Go to *http://www.alphaworks.ibm.com/tech/ettk* and download the install file *ettk10.exe*.

Once the install file is downloaded, run it with all the default options. This should install the ETTK in the directory *c:\ettk*. The rest of this tutorial assumes the ETTK is installed in the location *c:\ettk*, so please adjust accordingly.

Unlike the older version of the ETTK (called the Web Services ToolKit or WSTK), this version does not ship with a default application server which the install package needs. You will need to install Tomcat 4.0 (or one of the other approved application servers). To make the ETTK install happy please download and install Tomcat 4.0. Then point the ETTK install to the Tomcat 4.0 directory when it prompts you. Tomcat 4.0 can be found at *http://jakarta.apache.org/builds/jakarta-tomcat-4.0/release/v4.0.6/bin/* .

Since this tutorial uses EJB technology, you will need another application server that supports EJB components. The examples in this tutorial use Resin EE. However it should be easy for you to port to another application server. If you are not in the mood for porting, please download and install Resin EE 2.x at this time from *www.caucho.com*. Install Resin EE in *c:\resin*.

If you do not have Ant, now is a good time to install it as well. You can find Ant at

Apache.org.

If you do not have an IDE, please download and install Eclipse from the *Eclipse project site*.

Create a simple EJB component

Start your project by creating a directory on your hard drive called *C:\tutorials\axisEJB\axisEJBHello*.

In *axisEJBHello*, create a subdirectory called *src*. In the *src* directory create another subdirectory structure to hold the package structure *rickhightower.axis.ejb.tutorial*.

This EJB component is going to need an implementation, a remote interface, a home interface, and a deployment descriptor.

In the directory *C:\tutorials\axisEJB\axisEJBHello\src\rickhightower\axis\ejb\tutorial*, create an EJB Implementation class (*HelloService.java*) as follows:

```
/*
 */

package rickhightower.axis.ejb.tutorial;


import javax.ejb.SessionBean;
import javax.ejb.SessionContext;


/**
 * HelloService session bean.
 * @ejb.bean name="Hello" type="Stateless"
 *         local-jndi-name="Hello"
 */

public class HelloService implements SessionBean {
        /**
         * This method is part of the ejb interface.
         * @ejb.interface-method view-type="both"
         * @ejb.transaction type="Required"
         */
        public String[] getGreetings() {
                return new String[]{
                        "Hello cruel world!",
                        "Hello mom",
                        "Hello all"
                };
        }
```

```
        public void ejbActivate() {
        }

        public void ejbPassivate() {
        }

        public void ejbRemove() {
        }

        public void ejbCreate() {
        }

        public void setSessionContext(SessionContext sc) {
        }
}
```

Create a remote interface in the same directory as follows:

```
package rickhightower.axis.ejb.tutorial;


public interface Hello extends javax.ejb.EJBObject {
        public java.lang.String[] getGreetings()
                     throws java.rmi.RemoteException;
}
```

Create a home interface in the same directory as follows:

```
package rickhightower.axis.ejb.tutorial;

public interface HelloHome extends javax.ejb.EJBHome {
        public Hello create()
                throws javax.ejb.CreateException, java.rmi.RemoteException;
}
```

Create a deployment descriptor in the META-INF directory as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!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 Beans -->
```

```
      <session >
         <ejb-name>Hello</ejb-name>

         <home>rickhightower.axis.ejb.tutorial.HelloHome</home>
         <remote>rickhightower.axis.ejb.tutorial.Hello</remote>
         <ejb-class>rickhightower.axis.ejb.tutorial.HelloService</ejb-class>
         <session-type>Stateless</session-type>
         <transaction-type>Container</transaction-type>

      </session>
   </enterprise-beans>

   <!-- Assembly Descriptor -->
   <assembly-descriptor >

   <!-- transactions -->
           <container-transaction >
              <method >
                 <ejb-name>Hello</ejb-name>
                      <method-intf>Remote</method-intf>
                 <method-name>getGreetings</method-name>
                 <method-params>
                 </method-params>
              </method>
              <trans-attribute>Required</trans-attribute>
           </container-transaction>
   </assembly-descriptor>

</ejb-jar>
```

Create a Web application and configure *web.xml*

In *axisEJBHello* directory create a directory called *web* (*C:\tutorials\axisEJB\axisEJBHello\web*). Under *web* create a directory called *WEB-INF* (*C:\tutorials\axisEJB\axisEJBHello\web\WEB-INF*).

In *WEB-INF* create two directories: *lib* and *classes*. Copy the following files into *lib*:

- *C:\ettk\wsag\axis\lib\axis.jar*
- *C:\ettk\wsag\axis\lib\commons-discovery.jar*
- *C:\ettk\wsag\axis\lib\commons-logging.jar*
- *C:\ettk\wsag\axis\lib\jaxrpc.jar*
- *C:\ettk\wsag\axis\lib\log4j-1.2.4.jar*
- *C:\ettk\wsag\axis\lib\saaj.jar*
- *C:\ettk\wsag\lib\activation.jar*
- *C:\ettk\wsag\ogsa\impl\java\lib\mail.jar*

- *C:\ettk\wsag\lib\mailapi.jar*

Create a standard *web.xml* file and put it in the *WEB-INF* directory (*C:\tutorials\axisEJB\axisEJBHello\web\WEB-INF*).

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">
<web-app>

</web-app>
```

Add the Axis Servlet (SOAP transport) and mappings to the *web.xml* file as follows:

```
  <servlet>
    <servlet-name>AxisServlet</servlet-name>
    <display-name>Apache-Axis Servlet</display-name>
    <servlet-class>
        org.apache.axis.transport.http.AxisServlet
    </servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>AxisServlet</servlet-name>
    <url-pattern>/servlet/AxisServlet</url-pattern>
  </servlet-mapping>

  <servlet-mapping>
    <servlet-name>AxisServlet</servlet-name>
    <url-pattern>/services/*</url-pattern>
  </servlet-mapping>
```

The *AxisServlet* is the entry point to the Axis Engine for doing SOAP over HTTP.

Add an EJB reference to the *web.xml* file to the **HelloService** EJB component as follows:

```
    <ejb-ref >
        <ejb-ref-name>Hello</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <home>rickhightower.axis.ejb.tutorial.HelloHome</home>
        <remote>rickhightower.axis.ejb.tutorial.Hello</remote>
        <ejb-link>ejbclient.jar#Hello</ejb-link>
    </ejb-ref>
```

Assuming that the EJB component and the WAR files are deployed in the same EAR file, the above would add an EJB reference to the Hello EJB component. You need two more files for this example. Unzip the sample source, find *happy.jsp* and *index.html* in the *Hello* example directory (under *web*), and copy *happy.jsp* and *index.html* into the *web* directory (*C:\tutorials\axisEJB\axisEJBHello\web*). You will use these files to see a list of Web services deployed in your Web application and to see if everything is correctly setup.

Create Web Service Deployment Descriptor for EJB component

So far you have downloaded and installed the ETTK. Then you created a simple EJB component called *Hello*. Then you created a Web application. You configured the Web application's *web.xml* file to use Axis as the SOAP transport. You also configured the Web application to refer to your EJB component called *Hello*.

You need a way for Axis to know that you want to expose your EJB component as a Web service. You can accomplish this with an Axis Web Service Deployment Descriptor (WSDD). The WSDD needs the ability to state the following about your EJB component: What kind of service?, Where to look for the service?, What is the structure of your EJB component?.

Thus you need to state the following:

1) What Kind of service?

- This service is EJB-based

2) Where to look for the service?

- The JNDI Context factory class for your EJB application server
- The JNDI URL of your EJB application server
- The home JNDI name of your EJB component

3) What is the structure of your EJB component?

- The remote interface of the EJB component
- The home interface of the EJB component

All of the above questions are answered with the Web Service Deployment Descriptor

file. Create a directory called *deploy* in *C:\tutorials\axisEJB\axisEJBHello*. Create a file called *deploy.wsdd* and put it in the *deploy* directory with the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns="http://xml.apache.org/axis/wsdd/"
            xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

    <service name="RemoteHelloService" provider="java:EJB">
        <parameter name="beanJndiName" value="Hello"/>
        <parameter name="homeInterfaceName"
                   value="rickhightower.axis.ejb.tutorial.HelloHome"/>
        <parameter name="remoteInterfaceName"
                   value="rickhightower.axis.ejb.tutorial.Hello"/>
        <parameter name="allowedMethods" value="getGreetings"/>
        <parameter name="jndiURL"
                   value="http://localhost:8080/AxisEJB/hessian"/>
        <parameter name="jndiContextClass"
                   value="com.caucho.hessian.HessianContextFactory"/>
    </service>
</deployment>
```

The file can be broken down as follows:

1) What Kind of service?

- This service is EJB-based

```
...

    <service name="RemoteHelloService" provider="java:EJB">

...
```

Notice that the provider type of the service is `java:EJB` (for example, `provider="java:EJB"`). The provider type of `java:EJB` specifies that this is an EJB-based Web service. A provider is a Handler responsible for implementing the logic of the service.

In this case, you are specifying that you are using a provider that will delegate the actual logic development to an EJB component. The EJB provider is part of the standard Axis distribution. The EJB provider extends the RPC provider. If you have worked with Axis before you likely worked with the `java:RPC` (`org.apache.axis.providers.java.RPCProvider`). The RPC provider implements message processing by walking over RPCElements of the SOAP envelope body and invoking the appropriate methods on a plain old Java Object (POJO). The

EJB provider (`java:EJB`, `org.apache.axis.providers.java.EJBProvider`) extends the RPC provider by looking up the EJB component with JNDI, getting its home method, and invoking the no argument `create` method on its home interface; it then takes the object returned from the home and treats it like `java:RPC` treats the POJO.

Here is the class hierarchy for the EJB Provider.

```
java.lang.Object
  |
  +--org.apache.axis.handlers.BasicHandler
        |
        +--org.apache.axis.providers.BasicProvider
              |
              +--org.apache.axis.providers.java.JavaProvider
                    |
                    +--org.apache.axis.providers.java.RPCProvider
                          |
                          +--org.apache.axis.providers.java.EJBProvider
```

If you have a few minutes, download the Axis source code and look at the EJBProvider and the RPCProvider. (Note there are other providers for RMI, CORBA, COM, and the Bean Scripting Language or BSF.)

2) Where to look for the service?

- The JNDI Context factory class for your EJB application server

```
...
    <service ...>
        ...
        <parameter name="jndiContextClass"
                   value="com.caucho.hessian.HessianContextFactory"/>
        ...
    </service>
...
```

The `jndiContextClass` parameter specifies the JNDI Context factory class for you EJB server. The example uses the one for Resin EE. Note this will vary from EJB application server to EJB application server. For example, when using JBoss, `jndiContextClass` would be `org.jnp.interfaces.NamingContextFactory`. You need to find this value for your application server of choice.

2) Where to look for the service?

- The JNDI URL of your EJB application server

```
...
    <service ...>
        ...
        <parameter name="jndiURL"
                   value="http://localhost:8080/AxisEJB/hessian"/>
        ...
    </service>
...
```

The `jndiURL` parameter specifies the JNDI URL for your EJB server. The example uses the JNDI URL for Resin EE on my box, which I have configured to port 8080. Note this will vary from EJB application server to EJB application server and depends on how you have the application server configured. For example, when using JBoss, `jndiURL` could be *jnp://localhost:1099*. You need to find this value for your application server and configuration.

2) Where to look for the service?

- The home JNDI name of your EJB component

```
...
    <service ...>
        <parameter name="beanJndiName" value="Hello"/>
    </service>
...
```

The `beanJndiName` parameter specifies where you have the EJB component bound into application servers JNDI tree. In this example, you have the EJB component bound to *Hello* in the JNDI tree. How to configure where the EJB component is bound in JNDI is specific to your EJB server. It usually involves an application server-specific deployment descriptor.

3) What is the structure of your EJB component?

- The remote interface of the EJB component

```
...
    <service name="RemoteHelloService" provider="java:EJB">
        ...
        <parameter name="remoteInterfaceName"
                   value="rickhightower.axis.ejb.tutorial.Hello"/>
...
    </service>
```

```
...
```

The `remoteInterfaceName` parameter specifies the remote interface class.

3) What is the structure of your EJB component?

* The home interface of the EJB component

```
...
    <service name="RemoteHelloService" provider="java:EJB">
        ...
        <parameter name="homeInterfaceName"
                   value="rickhightower.axis.ejb.tutorial.HelloHome"/>
       ...
    </service>
...
```

The `homeInterfaceName` specifies the home interface class.

Use Axis Ant Tasks to Deploy EJB Web service

So far you have downloaded and installed the ETTK, created a simple EJB component called `Hello`, created a Web application that references your EJB component, configured the Web applications *web.xml* file to use Axis as the SOAP transport, and configured the Web application to refer to your `Hello` EJB component. Lastly you created an Axis Web Service Deployment Descriptor that describes how to connect to your EJB component with the built-in EJB Provider. Now you need to use this Web Service Deployment Descriptor.

This step assumes that you have deployed the EJB component and Web Application to your application server of choice. The Web application should be mapped into a context called AxisEJB; if not, adjust accordingly. (The Ant script that ships with the sample builds and deploys the EJB component and Web Application to Resin EE. You will need to change this Ant script for your application server.)

Next you need to use the Axis Ant tasks to deploy the Web service to your application server. In order to start using the Axis Ant task, you need to set up a class path in Ant as follows:

```
<path id="axis-lib">
```

```
     <fileset dir="/ettk/wsag/axis/lib">
         <include name="**/*.jar"/>
     </fileset>
     <fileset dir="/ettk/wsag/lib">
         <include name="wsdl4j.jar"/>
     </fileset>
</path>
```

The above assumes you have the ETTK installed in *c:\ettk*. Next you need to declare the Axis Ant tasks by using the classpath you defined called axis-lib (`id="axis-lib"`). To declare an external Ant task, you need to use the `taskdef` as follows:

```
    <taskdef resource="axis-tasks.properties"
             classpathref="axis-lib" />
```

Now that you have the Axis Ant tasks imported, you can start to use them. Deploy the EJB component as a Web service with the Web Service Deployment Descriptor that you just created by using the `axis-admin` task as follows:

```
<target name="deploy.wsdd" depends="clean.config.axis,compile">
    <axis-admin
        port="8080"
        hostname="localhost"
        failonerror="true"
        servletpath="AxisEJB/services/AdminService"
        debug="true"
        xmlfile="deploy/deploy.wsdd"
    />
</target>
```

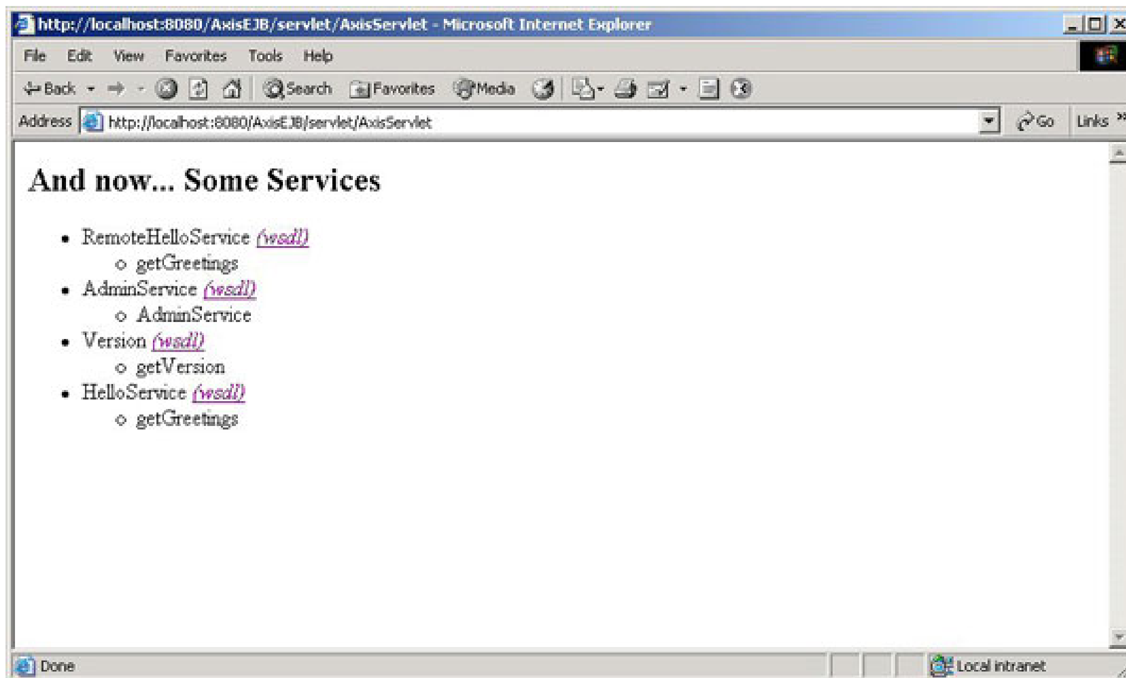
The `axis-admin` Ant task is used to administer a local or remote Axis server.

The above Ant target deploys the EJB component to a Web application at http://localhost:8080/AxisEJB by using the deployment descriptor you created earlier. Go ahead and run **deploy.wsdd target** at this time.

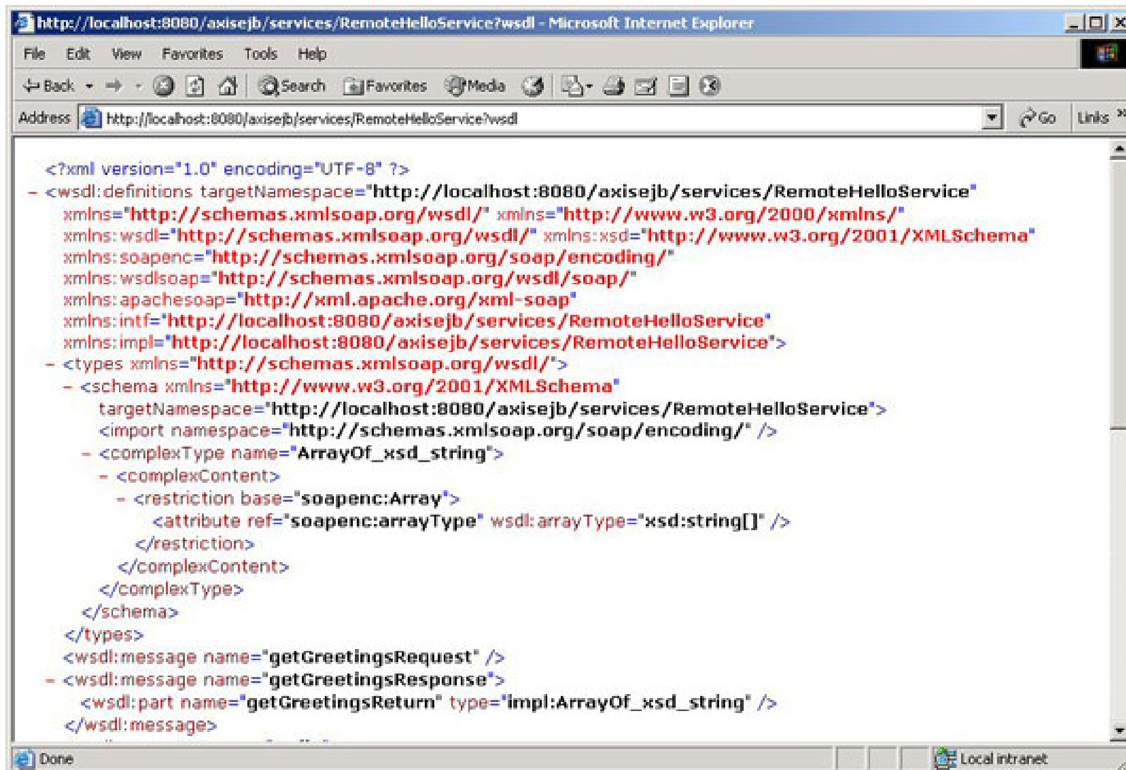
If the deployment fails, using a browser go to the *happyaxis.jsp* page (*http://localhost:8080/AxisEJB/happyaxis.jsp*), and see what is missing in your configuration. Make adjustments accordingly.

Next ensure that the deployment worked by going to *http://localhost:8080/AxisEJB/index.html* and clicking on the link that says **"View the**

list of deployed Web services". This link should pull up a page that looks like:



Make sure that *RemoteHelloService* is listed. Click on the **wsdl** link and you should get the following:



For more information on WSDL, see the IBM developerWorks tutorial "Web services with WSDL (WSDK)" (see Resources on page 31).

Use axis-wsdl2java task to generate client stubs

Now that you have developed and deployed your Web application, EJB component, and Web service, it is time to write a client. Before you write a client, generate the client stubs with the Axis `axis-wsdl2java` task as follows:

```
<target name="wsdl2java" >

    <axis-wsdl2java
      output="gen.client"
      testcase="true"
      verbose="true"
      url="http://localhost:8080/axisejb/services/RemoteHelloService?wsdl" >
    </axis-wsdl2java>

</target>
```

The `axis-wsdl2java` Ant Task creates Java classes from WSDL files. Notice that the `wsdl2java` target uses the `axis-wsdl2java` task to generate source code in the *gen.client* directory based on the WSDL file found at *http://localhost:8080/axisejb/services/RemoteHelloService?wsdl*. Since you also specified `testcase="true"`, not only will the `axis-wsdl2java` task create the stub code, it will create a JUnit test case client.

Here is what `axis-wsdl2java` task creates in this example.

```
C:\tutorials\axisEJB\axisEJBHello\gen.client>tree /a /f
C:.
\---localhost
    \---axisejb
        \---services
            |
            \---RemoteHelloService
                    Hello.java
                    RemoteHelloServiceSoapBindingStub.java
                    HelloService.java
                    HelloServiceLocator.java
                    HelloServiceTestCase.java
```

The Hello.java should look like the following:

```
/**
 * Hello.java
 *
 * This file was auto-generated from WSDL
 * by the Apache Axis WSDL2Java emitter.
 */

package localhost.axisejb.services.RemoteHelloService;
import java.rmi.*;

public interface Hello extends Remote {
    public String[] getGreetings() throws RemoteException;
}
```

The Hello interface follows the convention of JAX-RPC. Thus you will implement a JAX-RPC style client later. The **wsdl2java** also generates another interface called `HelloService`. (The code was modified to fit nicely on the screen.) This `HelloService` interface will be used for locating your Web service as follows:

```
/**
 * HelloService.java
 *
 * This file was auto-generated from WSDL
```

```
 * by the Apache Axis WSDL2Java emitter.
 */

package localhost.axisejb.services.RemoteHelloService;
import javax.xml.rpc.*;

public interface HelloService extends Service {
    public String getRemoteHelloServiceAddress();

    public Hello getRemoteHelloService() throws ServiceException;

    public Hello getRemoteHelloService(java.net.URL portAddress)
         throws ServiceException;
}
```

Notice that the `HelloService` has two methods for finding your service, both called `getRemoteHelloService()`. The no argument version of `getRemoteHelloService()` will return an instance of `Hello` interface for you to work for the service you have deployed at *http://localhost:8080/axisejb/services/RemoteHelloService*. (The code was modified to fit nicely on the screen).

The other version of `getRemoteHelloService()` that takes a URL is used to look up a particular Web service at another location. This is for when you deploy your Web service to production, for example, `getRemoteHelloService(new URL("http://www.rickhightower.com/AxisEJB"))`.

`RemoteHelloServiceSoapBindingStub` and `HelloServiceLocator` implement `Hello` and `HelloService` respectively. The class `HelloServiceTestCase` demonstrates how to use `RemoteHelloServiceSoapBindingStub` and `HelloServiceLocator` as follows:

```
/**
 * HelloServiceTestCase.java
 *
 * This file was auto-generated from WSDL
 * by the Apache Axis WSDL2Java emitter.
 */

package localhost.axisejb.services.RemoteHelloService;
import junit.framework.*;

public class HelloServiceTestCase extends TestCase {
    public HelloServiceTestCase(java.lang.String name) {
        super(name);
    }
    public void test1RemoteHelloServiceGetGreetings() throws Exception {
        RemoteHelloServiceSoapBindingStub binding;
        try {
            binding = (RemoteHelloServiceSoapBindingStub)
```

```
                    new HelloServiceLocator().getRemoteHelloService();
        }
        catch (javax.xml.rpc.ServiceException jre) {
            if(jre.getLinkedCause()!=null)
                jre.getLinkedCause().printStackTrace();
            throw new AssertionFailedError("JAX-RPC ServiceException caught: " + jre);
        }
        assertNotNull("binding is null", binding);

        // Time out after a minute
        binding.setTimeout(60000);

        // Test operation
        java.lang.String[] value = null;
        value = binding.getGreetings();
        // TBD - validate results
    }

}
```

You will use the above as an example to create your very own client.

Note that if you do not like the default packages that get created with `axis-wsdl2java`, you can use a nested mapping tag to map a namespace to a custom package name. For example:

```
<axis-wsdl2java
  output="gen.client"
  testcase="true"
  verbose="true"
  url="${local.wsdl}" >
    <mapping
      namespace="http://localhost:8080/ns/foobar"
      package="rick.foobar" />
  </axis-wsdl2java>
```

This would map the namespace `http://localhost:8080/ns/foobar` to package `rick.foobar`, which is much better than the default.

Write a Web service client for your EJB component

The code that Axis `wsdl2java` task generates is test code that is very specific to Axis. To create a more portable client, you should use reflection to dynamically load the `Service Locator` as follows:

```
import java.net.URL;

import localhost.axisejb.services.RemoteHelloService.Hello;
import localhost.axisejb.services.RemoteHelloService.HelloService;

public class HelloClient {

    public static final String serviceClass;
    public static final String url;

    static {
        /* Allow using other JAX-RPC implementations besides Axis */
        serviceClass = System.getProperty("hello.service",
        "localhost.axisejb.services.RemoteHelloService.HelloServiceLocator");

        /* Allow other locations for our Web service */
        url = System.getProperty("hello.url",
        "http://localhost:8080/axisejb/services/RemoteHelloService");

    }

    public static void main(String []args) throws Exception{
        Hello port; //The Hello interface, in Web Service this is known as a port
        HelloService service; //The service locator

        /*  Dynamically load the service locator. */
        service = (HelloService)
                Class.forName(serviceClass).newInstance();

        /*  Get the port from the service.
         *  The port is the hello service. */
        port = service.getRemoteHelloService(new URL(url));

        /* Get the greetings from the service */
        String[] greetings = port.getGreetings();

        /* Print out the greetings */

        for (int index=0; index < greetings.length; index++){
                System.out.println(greetings[index]);
        }

    }

}
```

There is another more complicated, less performant, more portable way to create a client. I will cover this later.

For more information on JAX-RPC see Resources on page 31 .

If you have been following along, please create a directory called *client* and create a file called *HelloClient.java* that implements a client based on the code listing above.

Working with Local EJB Services

The Axis EJB provider also works with local EJB components as well as remote EJB components. In order for the local EJB version to work, the EJB components have to be co-located with the Web Application. It's actually easier to work with Local EJB components since there are less parameters in the Axis Web Services Deployment Descriptor. Here is an example Axis Web Services Deployment Descriptor that works with a local EJB version of your EJB component.

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns="http://xml.apache.org/axis/wsdd/"
            xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

    <service name="HelloService" provider="java:EJB">
        <parameter name="beanJndiName" value="java:comp/env/ejb/Hello"/>
        <parameter name="homeInterfaceName"
            value="rickhightower.axis.ejb.tutorial.HelloLocalHome"/>
        <parameter name="remoteInterfaceName"
            value="rickhightower.axis.ejb.tutorial.HelloLocal"/>
        <parameter name="allowedMethods" value="getGreetings"/>
    </service>
</deployment>
```

Notice that you do not have to specify the JNDI entries for the JNDI context factory or the JNDI URL. You just have to specify where the local EJB component is mapped into the environment naming context (ENC) of the Web application. The Axis EJB provider will look up the local EJB component in the ENC, call the create method on the home, and invoke methods on the EJB just like the RPC provide does with a POJO.

Of course if you are going to use a Local EJB component, you need to define a local interface, a local home interface, add them to your *ejb-jar.xml* , and add an `ejb-local-ref` to your *web.xml* file.

Here is the `ejb-local-ref` you need to add to *web.xml* the following:

```
<ejb-local-ref >
    <ejb-ref-name>ejb/Hello</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <local-home>rickhightower.axis.ejb.tutorial.HelloLocalHome</local-home>
    <local>rickhightower.axis.ejb.tutorial.HelloLocal</local>
    <ejb-link>ejbclient.jar#Hello</ejb-link>
</ejb-local-ref>
```

The *Hello* example has both a local and remote version of the EJB Hello Web service. This includes an orthogonal set of Axis Ant tasks called in our Ant build file.

Section 3. Returning more complex types

HRSystem SessionBean and DeptValue Bean

I felt it would be disingenuous to create just one example that only returns simple types. Thus, I created a more complex example that returns a complex object, for example, a non-primitive type.

You can find a more complex example in the example source code under the directory *axisEJBDepartEmployees*. This example creates and exposes a stateless session bean called `HRSystem`. The stateless session bean is configured to work with two Entity beans that use CMP CMR 2.0.

Here is the code for HRSystem:

```
package ejb;

import bean.*;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

import java.util.*;


import util.*;

/**
 * Provides a session facade that works with cmp/cmr from EJB 2.0
 * based entity beans.
 *
 * @ejb.bean name="HRSystem" type="Stateless"
 *         local-jndi-name="HRSystem"
 *
 * @ejb.ejb-ref ejb-name="DeptBean" view-type="local"
 * @ejb.ejb-ref ejb-name="EmployeeBean" view-type="local"
 */
public class HRSystemBean implements SessionBean {

...
        /**
         * Get a list of all the depts.
         *
         * @return All the dept value objects.
         *
         * This method is part of the ejb interface.
         * @ejb.interface-method view-type="both"
         * @ejb.transaction type="Required"
         */
        public DeptValue[] getDepartments() {
                ArrayList deptList = new ArrayList(50);
```

```
                Collection collection = LocalFinderUtils.findAll("DeptBean");
                return collectionToDeptValueArray(collection);
        }

        private DeptValue[] collectionToDeptValueArray(Collection collection){
                        ArrayList deptList = new ArrayList(50);
                        Iterator iterator = collection.iterator();
                        System.out.println("GOT THIS FAR");
                        while(iterator.hasNext()){
                                try {
                                        Dept dept = (Dept)iterator.next();
                                        DeptValue deptValue = new DeptValue();
                                        deptValue.setId(dept.getId());
                                        deptValue.setName(dept.getName());
                                        deptList.add(deptValue);
                                }catch(Exception e){e.printStackTrace(System.out);}
                        }
                return (DeptValue[])
                                deptList.toArray(new DeptValue[deptList.size()]);

        }

        public void ejbActivate() {}
        public void ejbPassivate() {}
        public void ejbRemove() {}
        public void ejbCreate() {}
        public void setSessionContext(SessionContext sc) {}

}
```

Notice that the `getDepartments()` returns an array of `DeptValues`. The `DeptValue` is just a POJO. In fact, `DeptValue` is a JavaBean, which is defined as follows:

```
package bean;

public class DeptValue implements java.io.Serializable {

    private String name;
    private Integer id;

    public DeptValue() {
    }

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

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

    public Integer getId() {
```

```
        return this.id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

}
```

Declaring Type Mappings in the WSDD

The `HRSystem` Service has a similar Ant build script and structure to the last example. Except now you have to add custom serializers. Luckily Axis provides serializers for `Arrays` and `Beans`. This is exactly what you need since you have an Array of Beans (`DeptValue[]` to be exact). Notice that the additions to Axis Web Service Deployment Descriptor for the `DeptValue` are as follows:

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns="http://xml.apache.org/axis/wsdd/"
            xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

    <service name="HRService" provider="java:EJB">

      <!-- Remote EJB specific parameters  -->
      <parameter name="beanJndiName" value="HRSystem"/>
      <parameter name="homeInterfaceName" value="ejb.HRSystemHome"/>
      <parameter name="remoteInterfaceName" value="ejb.HRSystem"/>
      <parameter name="jndiURL"
                 value="http://localhost:8080/hrsys/hessian"/>
      <parameter name="jndiContextClass"
                 value="com.caucho.hessian.HessianContextFactory"/>

      <!-- Parameters for type mappings -->
      <parameter name="wsdlTargetNamespace"
                 value="http://localhost:8080/hrsys/services/HRService"/>
      <parameter name="wsdlServiceElement"
                 value="HRSystemService"/>
      <parameter name="wsdlServicePort"
                 value="HRService"/>
      <parameter name="wsdlPortType"
                 value="HRSystem"/>
      <parameter name="allowedMethods"
                 value="getDepartments"/>

      <!-- Type mappings  -->
      <typeMapping
```

```
          xmlns:ns="http://localhost:8080/hrsys/services/HRService"
          qname="ns:ArrayOf_tns1_DeptValue"
          type="java:bean.DeptValue[]"
          serializer="org.apache.axis.encoding.ser.ArraySerializerFactory"
          deserializer="org.apache.axis.encoding.ser.ArrayDeserializerFactory"
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        />
        <typeMapping
          xmlns:ns="http://bean"
          qname="ns:DeptValue"
          type="java:bean.DeptValue"
          serializer="org.apache.axis.encoding.ser.BeanSerializerFactory"
          deserializer="org.apache.axis.encoding.ser.BeanDeserializerFactory"
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        />


    </service>
</deployment>
```

Note that the `HRSystem` example also has a local EJB version of the service. The Ant script and deployment descriptor are all set to deploy to Resin EE. You can adapt them to work with your application server of choice.

When you ran **wsdl2java** with the `HRSystem` it generated a special version of `DeptValue` as follows:

```
/**
 * DeptValue.java
 *
 * This file was auto-generated from WSDL
 * by the Apache Axis WSDL2Java emitter.
 */

package bean;

public class DeptValue  implements java.io.Serializable {
    private java.lang.Integer id;
    private java.lang.String name;

    public DeptValue() {
    }

    public java.lang.Integer getId() {
        return id;
    }

    public void setId(java.lang.Integer id) {
        this.id = id;
    }

    public java.lang.String getName() {
        return name;
```

```
    }

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

    private java.lang.Object __equalsCalc = null;
    public synchronized boolean equals(java.lang.Object obj) {
        if (!(obj instanceof DeptValue)) return false;
        DeptValue other = (DeptValue) obj;
        if (obj == null) return false;
        if (this == obj) return true;
        if (__equalsCalc != null) {
            return (__equalsCalc == obj);
        }
        __equalsCalc = obj;
        boolean _equals;
        _equals = true &&
            ((this.id==null && other.getId()==null) ||
             (this.id!=null &&
              this.id.equals(other.getId()))) &&
            ((this.name==null && other.getName()==null) ||
             (this.name!=null &&
              this.name.equals(other.getName())));
        __equalsCalc = null;
        return _equals;
    }

    private boolean __hashCodeCalc = false;
    public synchronized int hashCode() {
        if (__hashCodeCalc) {
            return 0;
        }
        __hashCodeCalc = true;
        int _hashCode = 1;
        if (getId() != null) {
            _hashCode += getId().hashCode();
        }
        if (getName() != null) {
            _hashCode += getName().hashCode();
        }
        __hashCodeCalc = false;
        return _hashCode;
    }

    // Type metadata
    private static org.apache.axis.description.TypeDesc typeDesc =
        new org.apache.axis.description.TypeDesc(DeptValue.class);

    static {
        typeDesc.setXmlType(new javax.xml.namespace.QName("http://bean",
        "DeptValue"));
        org.apache.axis.description.ElementDesc elemField =
        new org.apache.axis.description.ElementDesc();
        elemField.setFieldName("id");
        elemField.setXmlName(new javax.xml.namespace.QName("", "id"));
        elemField.setXmlType(new javax.xml.namespace.QName
        ("http://schemas.xmlsoap.org/soap/encoding/", "int"));
```

```
        typeDesc.addFieldDesc(elemField);
        elemField = new org.apache.axis.description.ElementDesc();
        elemField.setFieldName("name");
        elemField.setXmlName(new javax.xml.namespace.QName("", "name"));
        elemField.setXmlType(new javax.xml.namespace.QName
        ("http://www.w3.org/2001/XMLSchema", "string"));
        typeDesc.addFieldDesc(elemField);
    }

    /**
     * Return type metadata object
     */
    public static org.apache.axis.description.TypeDesc getTypeDesc() {
        return typeDesc;
    }

    /**
     * Get Custom Serializer
     */
    public static org.apache.axis.encoding.Serializer getSerializer(
           java.lang.String mechType,
           java.lang.Class _javaType,
           javax.xml.namespace.QName _xmlType) {
        return
          new  org.apache.axis.encoding.ser.BeanSerializer(
            _javaType, _xmlType, typeDesc);
    }

    /**
     * Get Custom Deserializer
     */
    public static org.apache.axis.encoding.Deserializer getDeserializer(
           java.lang.String mechType,
           java.lang.Class _javaType,
           javax.xml.namespace.QName _xmlType) {
        return
          new  org.apache.axis.encoding.ser.BeanDeserializer(
            _javaType, _xmlType, typeDesc);
    }

}
```

Notice that this version builds and registers a custom serializer for the Java client to use. To keep this version separate from the one in the client, I created a new project called *axisEJBDeptClient* in the sample.

Creating a Portable client for the HRSystem service

This is the more portable, slower version of the client. It is slower because it has to read and parse the WSDL file when it runs.

```
import javax.xml.namespace.QName;
import javax.xml.rpc.Service;
import javax.xml.rpc.ServiceFactory;

import java.net.URL;

import localhost.hrsys.services.HRService.HRSystem;

import bean.*;

public class Client {
    public static void main(String[] args) throws Exception {

        URL urlWsdl =
            new URL("http://localhost:8080/hrsys/services/HRService?wsdl");
        /* These values were collected by inspecting the WSDL file */
        String nameSpaceUri = "http://localhost:8080/hrsys/services/HRService";
        String serviceName = "HRSystemService";
        String portName = "HRService";


        ServiceFactory serviceFactory = ServiceFactory.newInstance();

        Service service =
                serviceFactory.createService(
                urlWsdl,
                new QName(nameSpaceUri, serviceName));

        HRSystem myProxy =
                (HRSystem) service.getPort(
                new QName(nameSpaceUri, portName),
                HRSystem.class);

        DeptValue[] values = myProxy.getDepartments();

        for (int index=0; index< values.length; index++){
                System.out.println(values[index].getName());
        }
    }

}
```

To find more details on JAX-RPC and creating portable clients, see Resources on page 31
.

This client is a bit of a pain write because you have to find the namespace URI, service name, and portName by inspecting the WSDL file. I prefer the little bit less portable, much easier to write version of a JAX-RPC client that you created before for Hello.

Section 4. Summary & further resources

Summary

In this tutorial you downloaded and installed the ETTK. Then you created a simple EJB component called `Hello`. You learned how to configure a Web applications *web.xml* file to use Axis as the SOAP transport. You learned how to write an Axis Web Service Deployment Descriptor that uses the Axis EJB provider. Then you learned how to deploy your Web service using Axis Ant task `axis-admin`. Then you generated client stub code using the Axis ant task `axis-wsdl2java`, and used this to write a Web service client.

Then you went on to learn how to write a Web service that uses complex Java types. Last but not least, you learned how to create a portable client with JAX-RPC.

Resources

- Easily find the following downloads:
  - Find the *Resin EE* application server this tutorial used to test the source code.
  - Also find other J2EE-compliant application servers like *IBM WebSphere* or *JBoss*.
  - Ant can be found at the *Ant home page*.
  - The ETTK can be found at the *ETTK site*. The examples in this tutorial use version ETTK 1.0, which includes Axis 1.1 release candidate 2.
  - Find *Eclipse* at the Eclipse Web Site.
  - Download *WebSphere Studio Application Developer trial*.
  - You can find *Tomcat 4.0* at jakarta.apache.org.
  - You can download Ant at *Apache.org*.
- To find more details on JAX-RPC and creating portable clients see:
  - *"Introduction to Web services and the WSDK."* (*developerWorks*, February 2003)

    - *"Introduction to creating a Web service from a Java class."* (*developerWorks*, February 2003)
    - *"Web services with WSDL (WSDK)."* (*developerWorks*, February 2003)
    - *"Creating a Web service from a Stateless Session Bean (WSDK)."* (*developerWorks*, February 2003)

- Learn more about Ant, from Rick's *Mastering Tomcat on Developing Web Components with Ant*.
- Learn more from the tutorial, "*Enhance J2EE component reuse with XDoclets*." (*developerWorks*, May 2003)
- For more informaiton on WSDL, see the IBM developerWorks tutorial *"Web services with WSDL (WSDK)"*. (*developerWorks*, February 2003)
- Read these other tutorials that Rick has helped author:
    - *Introduction to EJB CMP and CMR 2.0 part 1 of 4*. (*developerWorks*, March 2002)
    - *Introduction to EJB CMP and CMR 2.0 part 2 of 4*. (*developerWorks*, March 2002)
    - *Introduction to EJB CMP and CMR 2.0 part 3 of 4*. (*developerWorks*, July 2002)
    - *Introduction to EJB CMP and CMR 2.0 part 4 of 4*. (*developerWorks*, July 2002)
    - *Enhance J2EE component reuse with XDoclets*. (*developerWorks*, May 2003)

Feedback

Please send us your feedback on this tutorial. We look forward to hearing from you!

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our

production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.