Use XDoclet to generate Web service support files

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.

Section 1. About this tutorial

What is XDoclet?

You can skip this page if you already use XDoclet or already read the first XDoclet tutorial.

XDoclet facilitates automated deployment descriptor generation. XDoclet, a code generation utility, allows you to tack on metadata to language features like classes, methods, and fields using what looks like JavaDoc tags. Then it uses that extra metadata to generate related files like deployment descriptor and source code. This concept has been coined attribute-oriented programming (not to be confused with aspect-oriented programming, the other AOP).

XDoclet generates these related files by parsing your source files similar to the way the JavaDoc engine parses your source to create JavaDoc documentation. In fact, earlier versions of XDoclet relied on JavaDoc. XDoclet, like JavaDoc, not only has access to these extra metadata that you tacked on in the form of JavaDoc tags to your code, but also access to the structure of your source, that is, packages, classes, methods, and fields. It then applies this hierarchy tree of data to templates. It uses all of this and templates that you can define to generate what would otherwise be monotonous support files. Unlike the last tutorial on XDoclet, this tutorial does not focus on using existing templates that ship with XDoclet. Instead, in this tutorial you will create your own custom templates and XDoclet subtasks.

XDoclet ships an Ant task that enables you to create web.xml files, ejb-jar.xml files, and much more. In this tutorial, you will use XDoclet to generate a Web application deployment descriptor with the webdoclet Ant task. In addition you will generate EJB support files. Note that XDoclet Ant tasks do not ship with the standard distribution of Ant. You will need to download the XDoclet Ant tasks from http://xdoclet.sourceforge.net.

So you may wonder: "Why should I care? I am an excellent Java/J2EE Web developer and I have never needed XDoclet". Or you may say: "I already use XDoclet, why do I need to write my own templates?" As I stated before, you don't know what you are missing. Once you start use XDoclet, you will not stop. Once you start writing your own templates you will never repeat yourself again. If you are writing dry, mundane code, then you could probably use XDoclet instead. Allow XDoclet to generate the monotonous stuff, and stick to writing the good stuff. Computers were invented to do monotonous stuff to free humans to do creative things. XDoclet frees developers from monotonous code. XDoclet is the missing piece in your J2EE and Web service development process. It will speed your development. You must master how to use XDoclet templates.

What do I need to know for this tutorial?

This tutorial assumes you have a working knowledge of Java programming language, and XML. Knowledge of Web services, J2EE 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 Web services, Ant, Java technology, J2EE, XML and Enterprise JavaBean (EJB) components are provided in the references section at the end of this tutorial. If you are not familiar with XDoclet, you should read the first XDoclet tutorial that I wrote for developerWorks which explains how to use XDoclet (see Resources on page 38).

Eclipse was used to create the examples. 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 Ant XDoclet 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*. Just read the sections on Ant development for now. Also if you are new to XDoclet please read the first tutorial in this series (see Resources on page 38). And lastly if you want more background on Axis EJB technology, please see the Axis EJB technology tutorial listed in the Resources on page 38 section.

About the author

Rick Hightower is a developer who enjoys working with Java technology, Ant and XDoclet. Rick is currently the CTO of *Trivera Technologies*, a global training, mentoring and consulting company focusing on enterprise development. Rick along with a worldwide group of Java ninjas, just updated a "Mastering Struts" course, which covers everything from how to write your own custom tags to mastering tiles and the validator framework. Rick is writing three chapters on Tiles, Validator Framework and Built in Action in the 2nd edition of James Goodwill's Mastering Struts book, to be published by John Wiley.

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. Rick will be working on the next edition in July 2003.

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

Rick just spoke at JavaOne (2003) on EJB CMP/CMR and XDoclet. Rick will also be speaking 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 Java Development Kit (JDK). All the examples in this tutorial use *Java 2 Platform, Standard Edition (J2SE SDK) 1.4.x*

All of the examples use Ant build scripts 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 XDoclet. XDoclet can be found at the *XDoclet site* . XDoclet, like Ant, is open source. The examples in this tutorial use version XDoclet 1.2 beta 2. Not only is it likely that XDoclet will be out of beta by the time you read this, but XDoclet has been recently accepted to be an Apache Jakarta project so if you do not find it at the above link, look for it at the *Apache Jakarta site*.

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

Section 2. Simple template to introduce XDoclet

Simple XDoclet template example

To get this tutorial started, let's kick it off with a simple template. Remember that XDoclet extends the idea of the JavaDoc engine to allow the generation of code and other files based on custom JavaDoc tags. Of course XDoclet also has access to the complete parse tree. Thus it has access to the class, its package structure, its methods, etc.

XDoclet provides its own template engine. The templating engine is similar in concept to JavaServer Pages (JSP) technology. Essentially it has two types of tags: block tags and content tags. Block tags control flow like `if`, and `for` statements in Java programming language. Content tags print out pieces of the current parse tree context, fpor example, class names, method names, parameters, etc.

I could talk about XDoclet all day, but if you are like me then you want me to "show the code." Without further ado, here is a simple template that looks for all EntityBeans and prints out their class name and the names of all of their cmp fields:

```
<XDtClass:forAllClasses type="javax.ejb.EntityBean">
  Classname=<XDtClass:className/>
  <XDtProperty:forAllPropertiesWithTag tagName="ejb.persistence">
       CMP Field = <XDtMethod:propertyName/>
  </XDtProperty:forAllPropertiesWithTag>
</XDtClass:forAllClasses>
```

This simple template demonstrates both block tags and content tags. The `forAllClasses` is an example of a block tag, it iterates over all of the classes that are passed to the template engine via a `fileset` in the ant build file that invokes XDoclet. The `forAllClasses` filters out the classes that are not of type javax.ejb.EntityBean using the `type` attribute (`type="javax.ejb.EntityBean"`).

The `className` is an example of a content tag, prints out the name of the current class.

The `forAllProperitesWithTag` is another example of a block tag. It iterates over all of the properties in the implementation class that have the XDoclet tag `@ejb.persistence` (these properties would relate to the cmp fields of this bean). Lastly, the `propertyName` is another example of a content tag as it displays the current property name in the iteration.

All of the other examples builds on these simple concepts, so let's break this down. The following figure shows the block tags in bold-black font and the contents tags in bold-red font. The blocks also have rounded rectangles to demonstrate their scope.

```
<XDtClass:forAllClasses  type="javax.ejb.EntityBean">

Classname >XDtClass:className/>

<XDtProperty:forAllPropertiesWithTag
             tagName="ejb.persistence">

           CMP Field = <XDtMethod:propertyName/>

</XDtProperty:forAllPropertiesWithTag>

</XDtClass:forAllClasses>
```

- Block tags (control flow tags)
  - forAllClass
  - forAllProperties
- Content tags
  - className
  - propertyName

The output of this code template for this project is as follows:

```
Classname=EmployeeBean
        CMP Field = id
        CMP Field = firstName
        CMP Field = lastName
        CMP Field = phone

 Classname=DeptBean
        CMP Field = id
        CMP Field = name
```

Section 3. First template: XDoclet architecture

Running Simple XDoclet template example

To run the template you would need the following code in your ant build file:

```
<target name="templatedoclet" >
   <taskdef
      name="templatedoclet"
      classname="xdoclet.DocletTask"
      classpathref="xdocpath"
   />
   <templatedoclet destdir="test">
      <fileset dir="${src}">
         <include name="**/*Bean.java"/>
      </fileset>
      <template
         templateFile="template/template.xdt"
         destinationfile="test.txt"/>
   </templatedoclet>
</target>
```

The `templatedoclet` task is used to execute a template. The `fileset` sub element is used to specify that you only want files in the src directory that end in "Bean.java". The template sub task is used to specify the template file being used and the destination file. The above example executes the template I defined earlier (template/template.xdt) and outputs the results to text.txt. See the following figure for more details.

To really understand how to develop custom XDoclet templates effectively, you should have an understanding of the XDoclet architecture -- so allow me to talk about it a bit before we delve further.

XDoclet architecture

XDoclet consists of three main components:

1. XJavaDoc Engine
2. XDoclet Engine
3. Modules

Modules consist of several pieces:

1. Tasks
2. Subtasks
3. Tag Handlers
4. Templates

XJavaDoc Engine -- An earlier version of XDoclet relied on the JavaDoc engine from Sun Microsystems. The XJavaDoc replaces the need for Sun's JavaDoc engine. The

XJavaDoc engine is five times faster than the JavaDoc engine and has extensions for decorating the parse tree with additional metadata using a simple API.

Like JavaDoc the XJavaDoc parses Java source files and builds a tree of information about the classes and language features (packages, methods, fields), and metadata. The XJavaDoc engine provides access through an easy to use API. The API provides the same information about a class as the JavaDoc API with some additional features associated with storing and reading metadata and other structures. XJavaDoc adds the ability to modify the JavaDoc tags at runtime. Thus, metadata can be inferred and defaulted to some reasonable value.

XDoclet Engine -- The XJavaDoc engine reads the tags, which makes up the metadata and the structure of the classes. The XDoclet engine consumes the information from the XJavaDoc engine to generate support files (source code, and deployment descriptors). The XDoclet provides a great templating engine that transforms a template to one or more support files. XDoclet has a module loader that dynamically loads XDoclet modules which are specified with the xdoclet.xml file contained in the module's jar file.

You do not need to create a module to create templates. Every top-level XDoclet Ant task has the ability to execute an arbitrary template in place of the template that ships with the module.

Modules Engine -- Modules consist of tasks, subtasks, tag handlers, and templates.

Subtasks -- A subtask specifies the default templates to invoke and allows you to pass configuration parameters to the template. You will write a subtask after you write a few templates. An example of a subtask is as follows:

```
<ejbdoclet ...>

     <localinterface/>

</ejbdoclet>
```

The `localinterface` above is a subtask under ejbdoclet. I will describe how to write another subtask under ejbdoclet to deal with exposing an EJB component as a Web service. You will learn how to implement subtasks and specify their structure and how to specify their relationship to their parents when I cover subtasks.

Templates -- The reason why XDoclet will not go away even after the metadata features of JDK 1.5 are released and in vogue, is its ability to use the metadata and do something useful with it, that is, its templating mechanism. Thus, future versions of XDoclet will be able to read both the JavaDoc-based metadata and the new metadata. Metadata is key to Web services because it can bridge the impedance mismatch between technologies like XML Schema and JavaBean components.

XDoclet templates generate both deployment descriptors and source code files. In fact the XDoclet templates can generate any type of file. It is this ability that really aids developing components with their myriads of configuration files and deployment descriptors. In fact, you may even write your own component frameworks and use XDoclet to generate additional deployment-type files.

Tag Handlers -- Tag Handlers are to XDoclet tags what Custom Tag handlers are to JSP Custom Tags. The XDoclet engine maps the tags to its corresponding taghandler by its name. Tag handler classes must subclass xdoclet.TagHandler . XDoclet uses reflection to invoke a method on the tag handler based on the name of the tag. Therefore, XDtProperty:forAllPropertiesWithTag is going to look for a method forAllPropertiesWithTag in the Property tag handler.

The template engine knows which tag handler to use by looking up the tag handler in the xdoclet.xml file that ships with the tags. Here is a snippet from the core xdoclet.xml file for the Property tag handler entry:

```
<taghandler

   <taghandler namespace="Property"

    class="xdoclet.tagshandler.PropertyTagsHandler"/>

...

/>
```

When the XDoclet encounters a XDtProperty:forAllPropertiesWithTag tag its going look for the method forAllPropertiesWithTag in the `xdoclet.tagshandler.PropertyTagsHandler` class. It does this by first stripping the XDt prefix off of `XDtProperty` and then looks up the mapping defined in the xdoclet.xml file.

Here is a partial listing of the Property tag handler:

```
public class PropertyTagsHandler extends AbstractProgramElementTagsHandler
{
...
    public void forAllPropertiesWithTag(String template, Properties attributes)
                                                          throws XDocletException
    {
        ...
        String requiredTag = attributes.getProperty("tagName");

        if (requiredTag == null) {

            throw new XDocletException(

            "missing required tag parameter in forAllPropertiesHavingTag");
        }

        XClass oldClass = getCurrentClass();

        XClass superclass = null;

        Collection already = new ArrayList();

        // loop over superclasses

        do {

            XMethod oldCurrentMethod = getCurrentMethod();
```

```
            Collection methods = getCurrentClass().getMethods();
            for (Iterator j = methods.iterator(); j.hasNext(); ) {
                XMethod currentMethod = (XMethod) j.next();
                log.debug("looking at method " + currentMethod.getName());
                if (currentMethod.getDoc().hasTag(requiredTag)) {
                    setCurrentMethod(currentMethod);
                    String propertyName = currentMethod.getPropertyName();
                    log.debug("property identified " + propertyName);
                    if (!already.contains(propertyName)) {
                        generate(template);
                        already.add(propertyName);
                    }
                }
                setCurrentMethod(oldCurrentMethod);
            }
            // Add super class info
            superclass = getCurrentClass().getSuperclass();
            if (superclass != null) {
                pushCurrentClass(superclass);
            }
        } while (superclass != null);
        setCurrentClass(oldClass);
    }
...
```

I won't go over the detail of creating your own custom tags in this tutorial. I will cover how to create your own subtasks and templates and leave the creation of your own XDoclet tag handlers to a future tutorial.

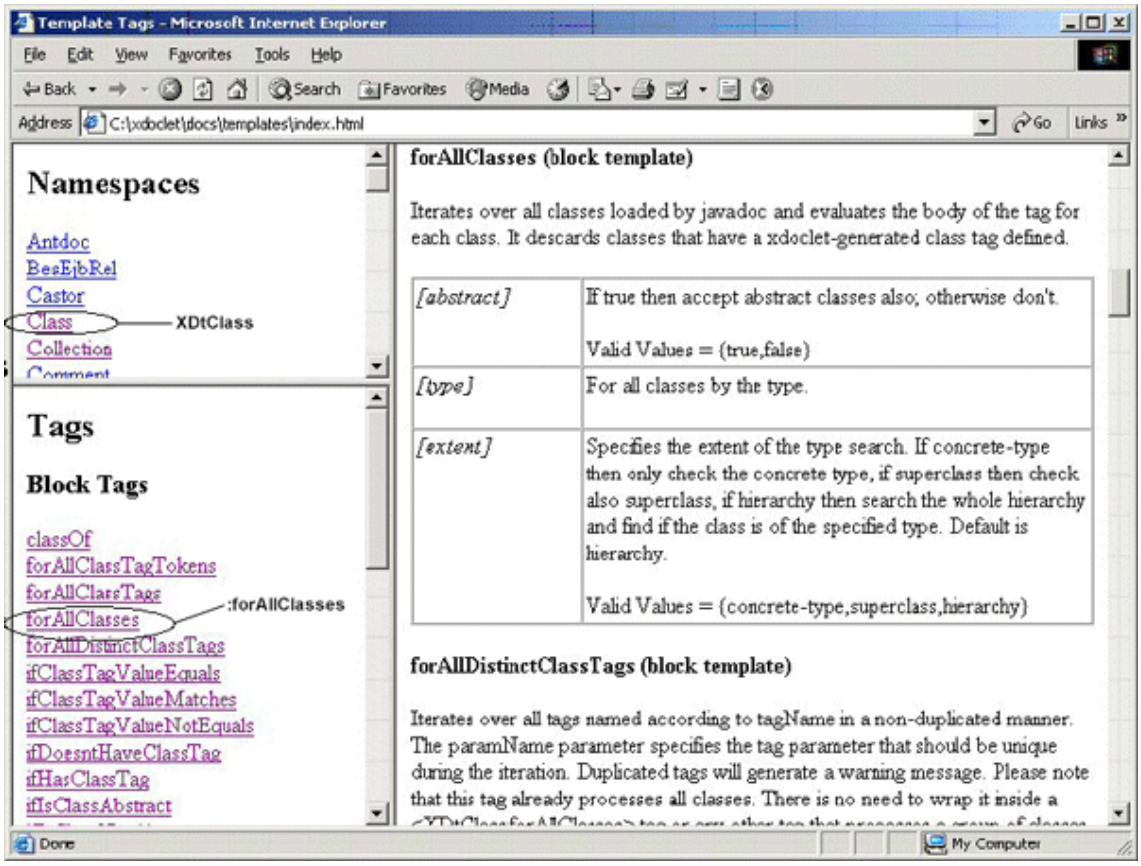
Now that you have a taste of the XDoclet architecture, I'll show you how to use your knowledge to develop your own custom tags that will generate an Axis Web services deployment descriptor.

Finding template block and content tags

So far I have covered a simple template, and then covered the XDoclet architecture. Now let me explain how you can develop your own custom templates. In order to develop your own custom templates you need to know which XDoclet block and content tags to use.

The best way to develop your own custom templates is to look at similar templates that XDoclet provides in its modules. You can do this by unjarring the modules and looking for files that end in xdt or downloading the XDoclet source and looking for the templates. In addition there is a way to look up all of the available template tags that ship with XDoclet via a documentation that resembles JavaDoc.

Luckily, XDoclet provides a reference to the entire template language. You can find this reference at [XDoclet Install Dir]\docs\templates\index.html. This is your guide to developing your very own custom templates.



If you are familiar with JavaDoc, using the template language reference is very easy. The XDoclet tag handlers correspond to the namespace of the tags and they show up in the top left panel. The actual tag for a particular handler shows up in the bottom left, and the documentation for the tag shows up in the right panel. If you want to look up a tag (for example, XDtClass:forAllClasses), strip off the Xdt (for example, XdtClass becomes class) from the namespace, and then look up the tag (for example, forAllClasses). See the above figure to see how to look up XDtClass:forAllClasses.

Now that you have the basics of XDoclet down, it's time to put it to use by creating an Axis EJB component Web Services Deployment Descriptor (deploy.wsdd).

Section 4. Case study: Create Web Service Deployment Descriptor 1st try

Create Web Service Deployment Descriptor for EJB component

This section will cover the details of the type of information you need to capture, and the next session will cover the tags that capture that information. Then I will cover the template that generates the support file based on the tags that you defined.

Axis provides a way to expose your EJB components as a Web service. Apache Axis does this with a Web Service Deployment Descriptor (WSDD). The WSDD states the following facts about your EJB comhponent: What kind of service?, Where to look for the service?, What is the structure of your EJB component?

Thus it expects the following information for an EJB component:

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

An example Web service deployment descriptor may look like the following:

```
<?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, that is, `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.

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 your EJB server. The example uses the one for Resin EE.

Note that this will vary from EJB application server to EJB application server. For example, JBoss `jndiContextClass` would be org.jnp.interfaces.NamingContextFactory.

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 computer, 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 for 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 the 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.

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 that sports the Axis servlet.

It is 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. Your template should take this into account.

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 component just like the RPC provider does with a POJO.

All of the above code is taken from the hello example that has both a local and remote version of the EJB Hello Web Service. You can find this full example in the Axis EJB tutorial (see Resources on page 38 section for more details).

Defining your own tags

In order to add the extra tags needed to capture the data in the Axis deployment descriptor, you will need to define your own JavaDoc tags. Notice the following JavaDoc section above the HRSystem bean class:

```
/**
 * 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="java:comp/env/ejb/HRSystem"
 *  jndi-name="HRSystem"
 *
 * @ejb.ejb-ref ejb-name="DeptBean" view-type="local"
 * @ejb.ejb-ref ejb-name="EmployeeBean" view-type="local"
 *
 * @ejb.home      generate="local" local-class="ejb.HRSystemLocalHome"
 * @ejb.interface generate="local" local-class="ejb.HRSystemLocal"
 *
 * @ejb.home      generate="remote" remote-class="ejb.HRSystemHome"
 * @ejb.interface generate="remote" remote-class="ejb.HRSystem"
 *
 *
 * @axis.ejb jndiURL="http://localhost:8080/hrsys/hessian"
 * @axis.ejb jndiContextClass="com.caucho.hessian.HessianContextFactory"
 * @axis.ejb allowedMethods="getDepartments"
 * @axis.ejb view-type="local"
 *
 */
public class HRSystemBean implements SessionBean {
...
```

HRSystemBean class is a Stateless Session bean. (The HRSystem bean class was covered in both the Axis EJB tutorial and the Speed J2EE development with XDoclet tutorial. Both tutorials are listed in the Resources on page 38 section.) Most of the tags above are ejb.xxx tags (ejb.home, ejb.interface, etc.). You also added new tags just to capture the data that you need for the Axis deployment descriptor. The tags that correspond to the tags for the Axis deployment descriptor are all named axis.ejb.

```
* @axis.ejb jndiURL="http://localhost:8080/hrsys/hessian"
* @axis.ejb jndiContextClass="com.caucho.hessian.HessianContextFactory"
* @axis.ejb allowedMethods="getDepartments"
* @axis.ejb view-type="local"
```

You can see that you use this tag several times, and each time you specify a new attribute. The jndiURL specifies the location of the JNDI server. The jndiContextClass specifies the name of the JNDI context factory, and lastly the allowedMethods attribute specifies the list of allowed methods. (I will show you how to refactor this later to work a little nicer).

Template that generates Axis deployment descriptor (1st try)

What you want to do is iterate through all of the classes that are passed to the template and if the current class is SessionBean and if the SessionBean is Stateless, then you want to process the axis.ejb JavaDoc tags to pull out the metadata to insert into the Axis deployment descriptor.

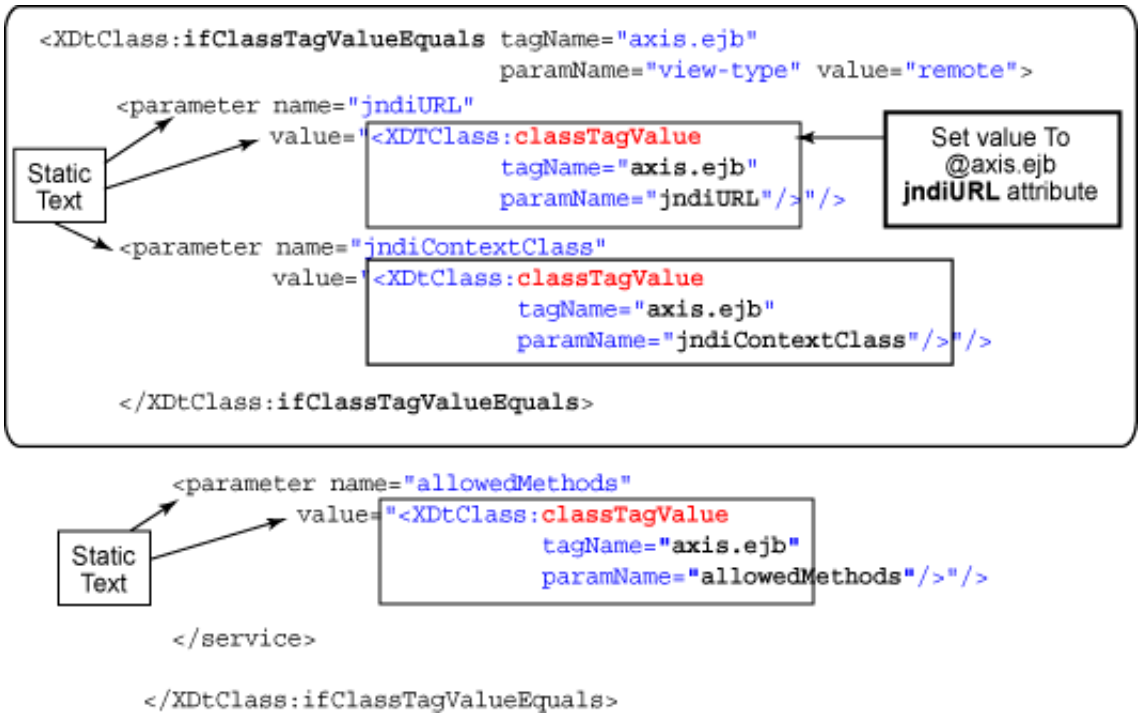
This template file will consist of static strings that represent the structure of the deployment descriptor, and then you will use block tags to see if certain tags and conditions are valid. If those tags and conditions are valid then you will output data from the tags in the source file to the new Axis deployment descriptor.

Let's first see the code to iterate over the current classes and see if they are Stateless Session beans as follows:

- Checks to see if the class is a SessionBean
- Checks to see if SessionBean is stateless

- Block tag
  - forAllClasses
  - ifClassTagValueEquals

Notice the mixing of static text and block tags. The next thing you need to do is process local and remote SessionBeans differently, and output values of the JavaDoc tags. This magic is done with the block tag `ifClassTagValuesEquals` and the content tag `classTagValue`. The following figure demonstrates this concept.



Notice how the tag `XDtClass:ifClassTagValueEquals` checks to see if the axis.ejb JavaDoc tags view-type parameter is set to remote. These techniques are

used throughout the template to output different values in the template.

Template listing that generates Axis deployment descriptor (1st try)

For completeness here is the entire template. Notice the combination of static and dynamic data from content tags:

```
<?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">
<XDtClass:forAllClasses type="javax.ejb.SessionBean">
<XDtClass:ifClassTagValueEquals tagName="ejb.bean"
                                paramName="type" value="Stateless">
<XDtClass:ifClassTagValueEquals tagName="axis.ejb"
                                paramName="view-type"
                                value="remote">
 <service name="<XDtClass:classTagValue
           tagName="ejb.bean"
           paramName="name"/>" provider="java:EJB">
    <parameter name="beanJndiName"
               value="<XDtClass:classTagValue
                               tagName="ejb.bean"
                               paramName="jndi-name"/>"/>
    <XDtClass:forAllClassTags tagName="ejb.home" >
    <XDtClass:ifClassTagValueEquals tagName="ejb.home"
                              paramName="generate"
                              value="remote">
    <parameter name="homeInterfaceName"
               value="<XDtClass:classTagValue
                               tagName="ejb.home"
                               paramName="remote-class"/>"/>
```

```
      </XDtClass:ifClassTagValueEquals>
      </XDtClass:forAllClassTags>
      <XDtClass:forAllClassTags tagName="ejb.interface" >
      <XDtClass:ifClassTagValueEquals tagName="ejb.interface"
                                   paramName="generate"
                                   value="remote">
      <parameter name="remoteInterfaceName"
                value="<XDtClass:classTagValue
                                   tagName="ejb.interface"
                                   paramName="remote-class"/>"/>
      </XDtClass:ifClassTagValueEquals>
      </XDtClass:forAllClassTags>
 </XDtClass:ifClassTagValueEquals>
 <XDtClass:ifClassTagValueEquals tagName="axis.ejb"
                                   paramName="view-type" value="local">
   <service name="<XDtClass:classTagValue
                              tagName="ejb.bean"
                              paramName="name"/>Local" provider="java:EJB">
      <parameter name="beanJndiName"
                value="<XDtClass:classTagValue
                                   tagName="ejb.bean"
                                   paramName="local-jndi-name"/>"/>
    <XDtClass:forAllClassTags tagName="ejb.home" >
     <XDtClass:ifClassTagValueEquals tagName="ejb.home"
                                 paramName="generate"
                                 value="local">
     <parameter name="homeInterfaceName"
                value="<XDtClass:classTagValue
                                   tagName="ejb.home"
                                   paramName="local-class"/>"/>
     </XDtClass:ifClassTagValueEquals>
     </XDtClass:forAllClassTags>
```

```
      <XDtClass:forAllClassTags tagName="ejb.interface" >
      <XDtClass:ifClassTagValueEquals tagName="ejb.interface"
                                      paramName="generate"
                                      value="local">
      <parameter name="remoteInterfaceName"
                 value="<XDtClass:classTagValue
                                      tagName="ejb.interface"
                                      paramName="local-class"/>"/>
      </XDtClass:ifClassTagValueEquals>
      </XDtClass:forAllClassTags>
  </XDtClass:ifClassTagValueEquals>
  <XDtClass:ifClassTagValueEquals tagName="axis.ejb"
                                      paramName="view-type" value="remote">
     <parameter name="jndiURL"
                 value="<XDtClass:classTagValue
                                      tagName="axis.ejb"
                                      paramName="jndiURL"/>"/>
     <parameter name="jndiContextClass"
                 value="<XDtClass:classTagValue
                                      tagName="axis.ejb"
                                      paramName="jndiContextClass"/>"/>
 </XDtClass:ifClassTagValueEquals>
     <parameter name="allowedMethods"
                 value="<XDtClass:classTagValue
                                      tagName="axis.ejb"
                                      paramName="allowedMethods"/>"/>
   </service>
 </XDtClass:ifClassTagValueEquals>
 </XDtClass:forAllClasses>
 </deployment>
```

Section 5. Case study: Create Web Service Deployment Descriptor 2nd try

Create Web Service Deployment Descriptor for EJB component

You can run the last template just like you ran the first template because it only relies on core XDoclet tags handlers. But, there are several things wrong with the last approach. Please do not misunderstand though; the last template is very useful and, more importantly, works. However all of the SessionBeans would be included in the same deployment descriptor. What if, during development you only want to redeploy one of the services, that is, the one you were working on?

The second problem with what I did above is that I only relied on the core tag handlers. I did not use the EJB specific tag handlers which make working with EJB technology easier. In order to use EJB specific tag handlers you would need to run your next template as a subtask of the EJB task.

The third issue is that I specified vendor specific configuration data in the deployment descriptor (jndiURL and jndiContextClass). I don't like having vendor specific configuration data in my classes, and neither should you. There are two ways around this. One is to use an ant property for the value as follows:

```
 ...
  * @axis.ejb jndiURL="${axis.jndiURL}"
  * @axis.ejb jndiContextClass="${axis.jndiContext}"
  */
 public class HRSystemBean implements SessionBean {
 ...
```

This is great, now I can specify the two above ant properties in my ant properties file and the value of jnidURL and jndiContextClass will use these values. But, I don't like this approach because I will need to add these same two lines of JavaDoc code in every one of my SessionBeans that I want to expose as a Web service in the entire project. This breaks the rule: "Don't repeat yourself." Instead of this let's pass in configuration parameters into your very own custom subtask as follows:

```
     <ejbdoclet
             ejbspec="2.0"
             mergeDir="${src}"
             destDir="${gen.src}"
                 >
```

```
            <fileset dir="${src}">
                  <include name="ejb/*Bean.java" />
            </fileset>
            <localinterface/>
            <localhomeinterface />
            <remoteinterface/>
            <homeinterface />
            <entitypk/>
            <apacheaxis destDir="./test"
                jndiURL="http://localhost:8080/hrsys/hessian"
                jndiContextClass="com.caucho.hessian.HessianContextFactory"
                validateXML="true"/>
            <deploymentdescriptor destdir="META-INF"
                                        destinationFile="ejb-jar.xml"
                                        validatexml="true" />
             ...
                </ejbdoclet>
    </target>
```

From the above, notice the new custom subtask that you are going to write as follows:

```
            ...
            <apacheaxis destDir="./test"
                jndiURL="http://localhost:8080/hrsys/hessian"
                jndiContextClass="com.caucho.hessian.HessianContextFactory"
                validateXML="true"/>
            ...
```

In the next section I will cover the steps needed to create your very own custom subtask that you can run like the above.

Steps to creating your own XDoclet SubTask

In order to create a custom subtask you will need to do the following:

1. Subclass xdoclet.XmlSubTask
2. Create getter and setter methods for configuration settings to pass to the template (jndiURL and jndiContextClass)
3. Validate the configuration parameters by overriding the validateOptions method
4. Use the inherited and new configuration parameters in the new template and the parent tag handlers (EJB tag handlers)
5. Create a module.xml file and state that this new module depends on the EJB module
6. Run the xdoclet-xml.xdt template against your new SubTask to generate the SubTask deployment descriptor
7. Compile and Jar your new subtask with its module.xml file and its xdoclet.xml file

---

Subclass xdoclet.XmlSubTask

Since you are going to be generating XML output you want to subclass the XmlSubTask.

The class hierarchy for the XDocletSubTask is as follows:

```
+--xdoclet.DocletSupport
       |
       +--xdoclet.SubTask
              |
              +--xdoclet.TemplateSubTask
                     |
                     +--xdoclet.XmlSubTask
```

Ant uses all setter methods to automatically set config parameters, expressed in XML as attributes to the SubTask. Let's go over the configuration parameters (expressed in JavaBean properties in the subtasks).

If you were developing a subtask that developed Java source code instead of XML, you would use TemplateSubTask. The XmlSubTask provides support for validating the output XML file. It has getter and setter methods for specifying the DTD, Public ID, Schema, System ID, XML Encoding for a particular XML file. The getter and setter methods allow configuration of these parameters in the template. See the javadocs for XmlSubTask for more detail ([xdoclet-install-dir]/docs/api/index.html).

The TemplateSubTask has getter and setter methods for specifying a template generation parameters as follows: destination file, type of class to generate from, the template file, the URL of the template, package substitution, and more. The TemplateSubTask operates per class mode or single output mode. The XmlSubTask subclasses the TemplateSubTask; thus inherits all of its attributes (properties). See the javadocs for TempalteSubTask for more detail ([xdoclet-install-dir]/docs/api/index.html).

TemplateSubTask subclasses the SubTask class, which is the abstract base class for all XDoclet subtasks. The SubTask specifies parameters like the merge directory, the destination directory and more. See the javadocs for SubTask for more detail ([xdoclet-install-dir]/docs/api/index.html).

The SubTask class subclasses the DocletSupport class. The DocletSupport class has methods that get called by the XDoclet framework to pass information about the parse tree. It sets the current package, the current class, the current tag, the current methods that are currently being processed. The XJavaDoc API is similar to the JavaDocs API (or even the Java reflection API) as it has objects that refer to classes, methods, packages, etc. In addition, the XJavaDoc API has classes that represent tags and attributes.

Here are some of the method signatures from the DocletSupport class that set the current object that can be worked on:

```
void setCurrentClass(xjavadoc.XClass clazz)

void setCurrentConstructor(xjavadoc.XConstructor constructor)

void setCurrentField(xjavadoc.XField field)

void setCurrentMethod(xjavadoc.XMethod method)

void setCurrentPackage(xjavadoc.XPackage pakkage)
```

All of the above setter methods have corresponding getter methods that can be used by the subclasses of DocletSupport. See the javadocs for DocletSupport for more detail ([xdoclet-install-dir]/docs/api/index.html).

With that, let's complete the first step by subclassing XmlSubTask as follows:

```
public class AxisSoapSubTask extends XmlSubTask {
        private final static String SOAP_SCHEMA =
                " http://xml.apache.org/axis/wsdd/";
        private static String DEFAULT_TEMPLATE_FILE =
                "resources/apache-axis-ejb-soap.xdt";
        private static String GENERATED_FILE_NAME = "axis-dds-{0}.xml";
         protected String ejbProvider = "java:EJB";
        protected Path providerClasspath;
        protected String jndiURL = null;
        protected String jndiContextClass = null;
        private static final String REMOTE = "remote";
        private static final String LOCAL = "local";
...
```

By subclassing XmlSubTask, AxisSoapSubTask inherits all of the configuration attributes of XmlSubTask, SubTask, and TemplateSubTask. The constructor of AxisSoapSubTask, your new subtask that generates Axis deployment descriptors, calls its super class configuration properties as follows:

```
public class AxisSoapSubTask extends XmlSubTask {

      ...
        public AxisSoapSubTask() {
                super.setTemplateURL(getClass().getResource(DEFAULT_TEMPLATE_FILE));
                super.setDestinationFile(GENERATED_FILE_NAME);
                super.setSchema(SOAP_SCHEMA);
                super.setHavingClassTag("axis.soap-service");
                super.setValidateXML(false);
        }
```

Thus you set the default template file, the default destination file name, and the default schema for the XML document you are generating. Notice that you also specify that you only want to process files that have the axis.soap-server tag, and you do not want the XML file you generate to be validated.

Notice that the generated file name is set to `axis-dds-{0}.xml`. The `{0}` will be replaced with the name of the current class. Therefore you are going to use a new template that just outputs one class at a time.

Create getter and setter methods for configuration settings

In addition to inheriting configuration settings from its super classes, the AxisSoapSubTask must create getter and setter methods for configuration settings that it would like to pass to the template (example: jndiURL and jndiContextClass).

Here are the configuration setting JavaBean properties of the AxisSoapSubTask:

```
public class AxisSoapSubTask extends XmlSubTask {

      ...
        /**
         * Gets the ejbProvider attribute of the AxisSoapSubTask object
         *
         * @return   The EjbProvider value
         */
        public String getEjbProvider() {
                return ejbProvider;
        }
```

```
  /**
   * Gets the ProviderClasspath attribute of
* the AxisSoapSubTask object
   *
   * @return   The ProviderClasspath value
   */
 public Path getProviderClasspath() {
         return providerClasspath;
}
   /**
   * Gets the JNDI URL attribute of the
* AxisSoapSubTask object
   *
   * @return   The JNDI URL value
   */
 public String getJndiUrl() {
         return jndiURL;
}
 /**
   * Gets the JNDI Context Factory class attribute of the
* AxisSoapSubTask object
   *
   * @return   JNDI Context Factory class
*/
 public String getJndiContextClass() {
         return jndiContextClass;
 }
 /**
   * Sets the ejbProvider attribute of the
* AxisSoapSubTask object
   *
   * @param ejbProvider  The new EjbProvider value
   */
```

```
    public void setEjbProvider(String ejbProvider) {
            this.ejbProvider = ejbProvider;
    }
     /**
      * Sets the JNDI URL attribute of the
    * AxisSoapSubTask object. The JNDI URL is the URL
      * associated with the JNDI context provider used
    * when looking up an EJB's home interface.
      *
      * @param url  The new ContextProviderUrl value
      */
    public void setJndiUrl(String url) {
            this.jndiURL = url;
    }
      /**
      * Sets the JNDI Context Factory class attribute
    * of the AxisSoapSubTask object.
    * The JNDI Context Factory class is the name of the
      * JNDI context factory used when looking up an EJB's
    * home interface.
      *
      * @param contextFactoryName  The new JNDI Context
    *                            Factory class value
      */
    public void setJndiContextClass(String contextFactoryName) {
            this.jndiContextClass = contextFactoryName;
    }
   ...
```

Be sure to read the comments of the above configuration settings, JavaBean properties.

Validate the configuration parameters

Typically subtasks validate the configuration parameters by overriding the

`validateOptions` method and checking to see if the required parameters are not null or just set to white space. For example, you might implement something that looks like this:

```
public void validateOptions() throws XDocletException {
    checkParam("jndiURL", this.getJndiUrl());
        checkParam("jnidContextClass", this.getJndiContextClass());
        super.validateOptions();
}
private void checkParam (String name, String value)
                            throws XDocletException{
 if (value == null) {
    throw new XDocletException(
    Translator.getString(
    XDocletTemplateMessages.class,
    XDocletTemplateMessages.TEMPLATE_PARAMETER_MISSING,
    new String[]{name}));
}
}
```

The above code checks to see if the jndiURL and the jndiContextClass are setup. The problem with this is that the jndiURL and the jndiContextClass configuration parameters are only needed if you are generating remote EJB components, that is, they are not needed for local EJB components that are collocated with the AxisEJB. In order to handle these on a per class level, you will need to override the engineStarted method as follows:

```
protected void engineStarted() throws XDocletException {
        XClass xclass   =   getCurrentClass();
        XDoc doc        =   xclass.getDoc();
        XTag tag        =   doc.getTag("axis.soap-service");
        String viewType =   tag.getAttributeValue("view-type");
        boolean remote  =   REMOTE.equals(viewType);
        if (remote){
                checkParam("jndiURL", this.getJndiUrl());
                checkParam("jnidContextClass", this.getJndiContextClass());
        }
...
```

Notice that you traverse the current class hierarchy to get a tag that specifies its view

type (which can be set to remote or local). This way you are only checking the jnidURL and jndiContextClass if its view type is set to remote as these are only needed for remote EJB components. The engineStarted method gets called each time a template is processed. Since you have set the generated file name to `axis-dds-{0}.xml` with the `{0}`, the template will be used against each class and the class name will replace the `{0}`.

Use configuration parameters in template and EJB module tags

Since your subtask inherits the configuration parameters from its superclasses, and defined new configuration parameters you can use them in the template. For example the template uses the XML encoding parameter from the XmlSubTask as follows (formatted to fit on page):

```
<?xml version="1.0" encoding=

"<XDtConfig:configParameterValue paramName='Xmlencoding'/>"?>
```

The template also uses the XML schema configuration parameters as well:

```
<deployment xmlns="<XDtXml:schema/>" ...
```

In addition to the above, you can use the configuration parameters that you setup in your subtask as follows:

```
...
           provider="<XDtConfig:configParameterValue
                                paramName='ejbProvider'/>">
...
    <parameter name="jndiURL"
               value="<XDtConfig:configParameterValue
                                paramName='jndiURL'/>"/>
    <parameter name="jndiContextClass"
               value="<XDtConfig:configParameterValue
                                paramName='jndiContextClass'/>"/>
```

Notice the use of XDtConfig:configParameterValue to pull the configuration settings out of your subtask.

Since the module subtask is a subtask of the EJB module, it can use all of its XDoclet tag handlers demonstrated as follows:

```
<!--
 <XDtType:ifIsOfType type="javax.ejb.SessionBean">
 Session Bean
     <XDtEjbSession:ifStatelessSession>
     Stateless
     EJB Provider = <XDtConfig:configParameterValue paramName='ejbProvider'/>
     EJB Home = <XDtEjbHome:homeInterface type='remote'/>
     </XDtEjbSession:ifStatelessSession>
  <XDtEjb:ifRemoteEjb>
   Remote JNDI = <XDtEjbHome:jndiName type='remote'/>
   JNDI URL = <XDtConfig:configParameterValue
                              paramName='jndiURL'/>
   JNDI Context Factory = <XDtConfig:configParameterValue
                              paramName='jndiContextClass'/>
  </XDtEjb:ifRemoteEjb>
  <XDtEjb:ifLocalEjb>
   Local JNDI <XDtEjbHome:jndiName type='local'/>
  </XDtEjb:ifLocalEjb>
 </XDtType:ifIsOfType>
 -->
```

If you compare the original template with the new template, you will notice it is terser because it uses the EJB module tags in the template.

Another improvement you make to the template is the ability to mark methods as methods that will be exposed in the Axis deployment descriptor. Thus in the Java source file for the HRService you specify the following method level tag:

```
/**...
* @axis.soap-service view-type="remote"
*/
public class AxisSoapSubTask extends XmlSubTask {
    ...
        /**
         * Get a list of all the depts.
```

```
         *
         * @return All the dept value objects.
         * ...
         * @axis.soap-method
         */
        public DeptValue[] getDepartments() {
                ArrayList deptList = new ArrayList(50);
                Collection collection = LocalFinderUtils.findAll("DeptBean");
                return collectionToDeptValueArray(collection);
        }
```

Notice the `getDepartments` is marked with `@axis.soap-method`. The template uses this to generate a list of methods that are going to be exposed with the following template code (modified to fit).

```
    <parameter name="allowedMethods"
               value="
     <XDtMethod:forAllMethods>
          <XDtMethod:ifHasMethodTag
                tagName='axis.soap-method'>
               <XDtMethod:methodName/>
          </XDtMethod:ifHasMethodTag>
     </XDtMethod:forAllMethods>"/>
```

Now your templates only need to specify two tags: one class tag (`@axis.soap-service`) and one method tag (`@axis.soap-method`).

Create XDoclet deployment, module files, and package the module

You need to do the following things to configure the subtask as an XDoclet module:

- Create a module.xml file and state that this new module depends on the EJB module
- Run the xdoclet-xml.xdt template against your new SubTask to generate the SubTask deployment descriptor
- Package the module with the above deployment descriptors

The module.xml file has to specify that this module depends on the EJB module as follows:

```
<?xml version="1.0"?>

<!DOCTYPE module

PUBLIC "-//XDoclet Team//DTD XDoclet Module 1.0//EN"
"http://xdoclet.sourceforge.net/dtds/xdoclet-module_1_0.dtd">

 <module>

        <module-dependency module-name="ejb"/>

</module>
```

The generated xdoclet.xml file looks as follows (modified to fit):

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

<!--

<!DOCTYPE xdoclet-module PUBLIC "-//XDoclet Team//DTD XDoclet Module 1.0//EN"
 "http://xdoclet.sourceforge.net/dtd/xdoclet-module_1_0.dtd">

-->

<xdoclet-module>

    <subtask

        name="apacheaxis"

        implementation-class="xdoclet.<...>.AxisSoapSubTask"

        parent-task-class="xdoclet.modules.ejb.EjbDocletTask"

    />

</xdoclet-module>
```

Notice that the xdoclet.xml file specifies that your subtask is a subtask of the parent task EjdDoclet.

Now in order to build, package, and deploy you use ant as follows:

```
<target name="xdocletgen">

    <echo>

    ------------------------------------------------

    - This is used to generate XDoclet.xml file

    -

    -
```

```
 ------------------------------------------------
 </echo>
 <taskdef name="xdoclet" classname="xdoclet.DocletTask"
          classpathref="xdocpath" />
 <xdoclet destdir="xdoc.META" verbose="true">
     <fileset dir="xdoclet.src"/>
     <template
         templateFile="${xdoclet.src.home}/modules/xdoclet-xml.xdt"
         destinationFile="xdoclet.xml" />
 </xdoclet>
<mkdir dir="modules.dest" />
<javac destdir="modules.dest" srcdir="xdoclet.src"
       classpathref="xdocpath" debug="true"/>
<mkdir dir="modules.dest/META-INF" />
<copy todir="modules.dest/META-INF" >
     <fileset dir="xdoc.META" >
      <include name="xdoclet.xml"/>
      <include name="module.xml"/>
     </fileset>
</copy>
<copy todir="modules.dest" >
     <fileset dir="xdoc.META/resources" >
      <include name="**/*.properties"/>
     </fileset>
</copy>
<copy todir="modules.dest/xdoclet/modules/apache/axis/soap/ejb">
    <fileset dir="xdoclet.src">
     <include name="**/*.xdt"/>
    </fileset>
</copy>
<jar jarfile="c:\xdoclet\lib\xdoclet-axis-ejb-module.jar">
     <fileset dir="modules.dest" />
```

```
        <metainf dir="xdoc.META">
         <include name="xdoclet.xml"/>
         <include name="module.xml"/>
        </metainf>
    </jar>
 </target>
```

Go ahead and download the code for the new template and subtask and review it.

Section 6. Summary and resources

Summary

This tutorial showed J2EE developers how to use XDoclet to write their own custom templates and subtask.

XDoclet enables simplified continuous integration and refactoring with component-oriented development using attribute-oriented programming. XDoclet allows you to radically reduce development time, by generating deployment descriptors and support code, allowing you to focus on application logic code. Not only can you use the plethora of templates that ship with XDoclet, but you can create your own. In addition you can create a subtask to pass in custom configuration parameters that you do not want to show up in the source files.

Resources

Download all the source code in the source code zipfile.

If you are new to EJB technology:

- Check out Brett McLaughlin's *EJB best practices column* on developerWorks Java technology zone.
- Take the first tutorial in a series of five on *Introduction to container-managed persistence and relationships* (example code uses XDoclet) by Rick Hightower (*developerWorks*, March 2002)
- *The Developer's Guide to Understanding EJB 2.0 (gain deeper understanding of the specification)*

If you are new to Custom Tags:

- *JSP taglibs: Better usability by design*, by Noel J. Bergman (*developerWorks*, December 2001)
- *J2EE Tutorial: Custom Tag tutorial*

If you want more detail about Ant:

- *Ant Primer*
- *Book: Java Tools for Extreme Programming covers Ant*

If you want to learn how to work with Struts and XDoclet:

- *Mastering Struts (Struts tutorial that uses XDoclet)*

If you want a more complex EJB example (for example, many to many relationship, and primary key classes) ported to JBoss, Resin and others EJB servers get the source code for the 5 part series on using EJB technology with CMP/CMR 2.0:

- *More XDoclet examples*

Previous tutorials on *developerworks*:

- *Enhance J2EE Component Reuse with XDoclets* (*developerWorks*, May 2003)
- *Service-enable EJB SessionBeans with the IBM Emerging Technologies Toolkit (ETTK)* (*developerWorks*, May 2003)

Feedback

Please let us know whether this tutorial was helpful to you and how we could make it better. We'd also like to hear about other tutorial topics you'd like to see covered.

For questions about the content of this tutorial, contact the author, Rick Hightower, at *Rick_M_Hightower@hotmail.com*.

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.