Enhance J2EE component reuse with XDoclets

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

Purpose of this tutorial

This tutorial shows J2EE developers how to use XDoclet to speed development. XDoclet simplifies continuous integration between components using attribute-oriented programming. It allows you to radically reduce development time by generating deployment descriptors and support code, allowing you to focus on application logic code.

If you are a J2EE development veteran, then you realize keeping code in sync with deployment descriptors can be a drag. Often you may need to reuse components with other applications or in other environments like other application servers or with other database systems. You need to keep separate deployment descriptor for each application/environment combination, even if only one or two lines of the large deployment descriptor changes, you need to have a deployment descriptor for every possible configuration. This can really slow down development. At times you may feel you spend more time syncing deployment descriptors than writing code.

XDoclet facilitates automated deployment descriptor generation. As a code generation utility, it allows you to tack on metadata to language features like classes, method, 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 manual creation of support files. This tutorial focuses on using existing templates that ship with XDoclet.

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 Enterprise JavaBeans (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 *XDoclet site on Sourceforge.net*.

This tutorial is a hands-on approach to learning how to use XDoclet to do J2EE component development. By the end of this tutorial, you will build several J2EE components using XDoclet. You will build a Servlet, a Custom Tag (taglib) and develop 3 EJB components.

You may be wondering: "Why should I care? I am an excellent Java/J2EE Web developer and I have never needed XDoclet." Simply put, you don't know what you are

missing. Once you start using XDoclet, you will not stop. XDoclet is the missing piece in you J2EE development process. It will speed development. Once you have mastered the basics, you can go on to generate your code based on your own custom XDoclet templates.

What do I need to know for this tutorial?

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

The source code in the tutorial has been tested with *Tomcat*, and *Resin EE*. The applications should be easy to port to other J2EE-compliant application servers like *IBM WebSphere Application Server*.

I used the Eclipse Framework 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 (written by yours truly) *"http://www.rickhightower.com/AntPrimer.pdf"*. Just read the sections on Ant development for now.

What this tutorial covers

This tutorial covers getting started with XDoclet to speed J2EE development. The tutorial has three step-by-step examples applying XDoclet development to Servlets, Custom Tags, and EJB. All examples, ships 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 programming language, Ant and XDoclet. Rick is currently the CTO of *Trivera Technologies*, a global training, mentoring and consulting company focusing 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.

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 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 the examples in this tutorial use J2SE SDK 1.4.1.

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.

You will, of course, need XDoclet itself which 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 a Apache Jakarta project so if you do not find it at the above link look for it at the *Apache Jakarta site*.

I recommend that you use an Integrated development environment (IDE) such as available from the Eclipse project, since there are quite a few jar files to manage. All the examples ship with the projects done in the freely available Eclipse IDE and are compatible with Eclipse and WebSphere Studio Application Developer (WebSphere Studio). 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 (WebSphere Studio) is not required, but can be found at the *Eclipse Web Site* and at *WebSphere Studio trial download* respectively. There is no requirement to use Eclipse, but the Eclipse project files are provided as a convenience to Eclipse and WebSphere Studio users. WebSphere Studio builds on top of Eclipse. Eclipse was used to build the sample applications.

Section 2. Step-by-step Servlet example

Servlet XDoclet example

To get this tutorial started, let's kick it off with a simple Servlet and XDoclet combination. Remember that XDoclet extends the idea of the JavaDoc engine to allow the generation of code and other files based on custom JavaDoc tags. XDoclet ships with an Ant task that enables you to create *web.xml* files, ejb-jar.xml files, and much more. In this section, you will use XDoclet to generate a Web application deployment descriptor with the **webdoclet** Ant task. This will be the simplest endeavor in this tutorial. Note that XDoclet Ant tasks do not ship with the standard distribution of Ant.

If you have worked with J2EE technology before, you know what the *web.xml* file is used to configure Web applications. The *web.xml* file is the deployment descriptor for the Web application. XDoclet allows you to generate the *web.xml* deployment descriptor using JavaDoc like tags within Servlet's source code.

The following Servlet specifies XDoclet tags that will be used to generate a *web.xml* file. Let's do a quick preview and then I will break down how these tags map to elements that are generated in the Web application deployment descriptor (*web.xml*).

```
/*
 * BasicServlet.java
 *
 */

package rickhightower.servlet;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.sql.*;
import java.sql.*;
import javax.naming.*;


/**
 *
 * @author  Rick Hightower
 *
 * @version 1.0
 * @web.servlet name="BasicServlet"
                display-name="Basic Servlet"
                load-on-startup="1"
 * @web.servlet-init-param name="hi" value="Ant is cool!"
 * @web.servlet-init-param name="bye" value="XDoc Rocks!"
 * @web.resource-ref description="JDBC resource"
 *                   name="jdbc/mydb"
 *                   type="javax.sql.DataSource"
 *                   auth="Container"
 * @web.servlet-mapping url-pattern="/Basic/*"
 * @web.servlet-mapping url-pattern="*.Basic"
 * @web.servlet-mapping url-pattern="/BasicServlet"
 */
public class BasicServlet extends HttpServlet {

    /** Initializes the servlet.
```

```
     */
    public void init(ServletConfig config) throws ServletException {
        super.init(config);

    }

    /** Destroys the servlet.
     */
    public void destroy() {

    }

    /** Processes requests for both HTTP GET and POST methods.
     * @param request servlet request
     * @param response servlet response
     */
    protected void processRequest(HttpServletRequest request,
                                  HttpServletResponse response)
                                  throws ServletException, java.io.IOException {
        ServletConfig config = this.getServletConfig();
        String hi = config.getInitParameter("hi");
        String bye = config.getInitParameter("bye");

        try{
            response.setContentType("text/html");

            java.io.PrintWriter out = response.getWriter();
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Basic Servlet</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1> bye:" + bye + "</h1>");
            out.println("<h1> hi:" + hi + "</h1>");
            getJdbcPool(out);
            out.println("</body>");
            out.println("</html>");
            out.close();
        }catch(Exception e){
            throw new ServletException(e);
        }
    }

    /** Handles the HTTP GET method.
     * @param request servlet request
     * @param response servlet response
     */
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
                         throws ServletException, java.io.IOException {
        processRequest(request, response);
    }

    /** Handles the HTTP POST method.
     * @param request servlet request
     * @param response servlet response
     */
    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response)
                          throws ServletException, java.io.IOException {
        processRequest(request, response);
    }

    /** Returns a short description of the servlet.
```

```
     */
    public String getServletInfo() {
        return "XDoc Rules";
    }

    private void getJdbcPool(java.io.PrintWriter out)throws Exception{
        out.println("</ br>");

        Object obj = new InitialContext().
                            lookup("java:comp/env/jdbc/mydb");
        DataSource pool = (DataSource)obj;
        if (pool == null) return;
        Connection connection = pool.getConnection();

        out.println("<table>");
        try{

            ResultSet rs =
                connection.getMetaData().
                getTables(null,null,null,null);
            while(rs.next()){
                out.println("<table-row><table-cell>");
                out.println(rs.getString("TABLE_NAME"));
            }
        }finally{

            connection.close();
        }
        out.println("</table>");



        out.println("</ br>");
    }

}
```

When you apply the XDoclet Ant task, **webdoclet**, to the above source file you will get the following deployment descriptor.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>

...

    <servlet>
       <servlet-name>BasicServlet</servlet-name>
       <display-name>Basic Servlet</display-name>
     <servlet-class>rickhightower.servlet.BasicServlet</servlet-class>

       <init-param>
          <param-name>hi</param-name>
          <param-value>Ant is cool!</param-value>
       </init-param>
       <init-param>
          <param-name>bye</param-name>
          <param-value>XDoc Rocks!</param-value>
       </init-param>
```

```
        <load-on-startup>1</load-on-startup>

    </servlet>

    <servlet-mapping>
        <servlet-name>BasicServlet</servlet-name>
        <url-pattern>/Basic/*</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>BasicServlet</servlet-name>
        <url-pattern>*.Basic</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>BasicServlet</servlet-name>
        <url-pattern>/BasicServlet</url-pattern>
    </servlet-mapping>

 ...

    <resource-ref>
        <description>JDBC resource</description>
        <res-ref-name>jdbc/mydb</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
    </resource-ref>

 ...

 </web-app>
```

The next few pages will explain step-by-step what tags correspond to which parts of
the Web application deployment descriptor.

Step one: define Servlet element

XDoclet may seem intimidating, but the mappings are quite natural. The first step in
using **webdoclet** is defining the servlet element as XDoclet JavaDoc tags at the class
level in your class as follows:

```
 ...
 * @web.servlet name="BasicServlet"
               display-name="Basic Servlet"
               load-on-startup="1"
 ...

 */

 public class BasicServlet extends HttpServlet {
```

This code generates the following servlet element and subelements in the *web.xml* file:

```
    <servlet>
       <servlet-name>BasicServlet</servlet-name>
       <display-name>Basic Servlet</display-name>
      <servlet-class>rickhightower.servlet.BasicServlet</servlet-class>

 ...

       <load-on-startup>1</load-on-startup>

    </servlet>
```

You may wonder how the `servlet-class` was determined. Since the XDoclet task works like the JavaDoc API, it can get the full classname of the servlet just like the JavaDoc API get the full classname for the JavaDocs. Not only does this save typing, it mitigates the likelihood of making mistakes. And then later when you are refactoring and decide to change the class name or package structure, you don't have to manually change all of the deployment descriptors. Whew!

Step two: define init parameters for Servlet

After the servlet is defined using the Servlet element, then you can define mappings and initial parameters. `servlet-init-param`s are defined in JavaDocs comments like this:

```
 ...
  * @web.servlet-init-param name="hi" value="Ant is cool!"
  * @web.servlet-init-param name="bye" value="XDoc Rocks!"
 ...
 */
 public class BasicServlet extends HttpServlet {
```

These parameters will generate the following >`init-param`>s in the deployment descriptor:

```
    <servlet>

       <servlet-name>BasicServlet</servlet-name>
 ...

       <init-param>
          <param-name>hi</param-name>
          <param-value>Ant is cool!</param-value>
       </init-param>
       <init-param>
          <param-name>bye</param-name>
          <param-value>XDoc Rocks!</param-value>
       </init-param>
 ...
    </servlet>
```

Hopefully you are looking at this example and screaming WAIT! Why Wait? I just hard code init parameters into the source code. Does this give you shivers? No? It should make you wonder. I was giving a talk at a conference about XDoclet and someone stopped me at this point and read me the riot act about how the `<init-param>`s were now hard coded. Not so fast ... Read on.

Step three: combine Ant and XDoclet to configure components

Hard coding initialization parameters into source code is something that you typically *do not* want to do. The whole idea around having `<init-param>`s is so the Web component can be customized by the application assembler into a J2EE application.

A better way is to set the init parameters to point to a token as in @bye@ and @hi@, and then later use the Ant copying with filtering enabled to pass the right token value for the right application. This assumes you are using Ant to build your project.

You can use Ant filtering to replace tokens in a configuration file with their proper values for the deployment environment. Filters are another way to support configuring J2EE components for more than one application. Here is an example Ant script: that configures the *web.xml* based on a condition:

```
<project name="filtering" default="run">

  <target name="spanishSetup" if="spanish">
      <filter token="bye" value="adios"/>
      <filter token="hi" value="hola"/>
  </target>

  <target name="englishSetup" unless="spanish">
      <filter token="bye" value="goodbye"/>
       <filter token="hi" value="hello"/>
  </target>

  <target name="setup" depends="spanishSetup,englishSetup"/>

  <target name="run" depends="setup">
      <copy todir="${workspace}/WEB-INF" filtering="true">
          <fileset dir="./WEB-INF"/>
      </copy>
  </target>
</project>
```

In above Ant example, the filter in the `englishSetup` target sets the bye token to *goodbye*, while the filter in the `spanishSetup` target sets the bye token to *adios*.

Later, when the script uses a copy task with filtering on, it applies the filter to all files in the file set specified by the copy. The copy task with filtering on replaces all occurrences of the string @bye@ with *adios* if the *spanish* property is set but to *later* if the *spanish* property is not set.

Even Easier way...

While this is one way to solve this problem, there is an easier way. Every attribute value in XDoclet can be set with an Ant Property. Since you generate the related files with XDoclet Ant tasks, XDoclet has access to all of the ant properties. Thus, you could set the values as follows:

```
...
 * @web.servlet-init-param      name="hi"
 *                      value="${basic.servlet.hi}"
 *
 * @web.servlet-init-param      name="bye"
 *                      value="${basic.servlet.bye}"
...
*/
public class BasicServlet extends HttpServlet {
```

Then when you generate the *web.xml* file the `hi` and `bye` initialization parameters would be set to whatever the current value of the `basic.servlet.hi` and `basic.servlet.bye` properties are set to in the Ant build script. XDoclet and Ant work well together to configure J2EE components into applications. The key take away is that the components can be configured on a per application basis. Imagine a master build file per application that calls build files per component passing the build files the info it needs to configure the component for that particular application.

Step four: define Servlet mappings

The servlet mappings can also be defined with the XDoclet JavaDoc tags as follows:

```
 * @web.servlet-mapping url-pattern="/Basic/*"
 * @web.servlet-mapping url-pattern="*.Basic"
 * @web.servlet-mapping url-pattern="/BasicServlet"
...
*/
public class BasicServlet extends HttpServlet {
```

These would generate the following entries in the *web.xml* file:

```
    <servlet-mapping>
       <servlet-name>BasicServlet</servlet-name>

       <url-pattern>/Basic/*</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
       <servlet-name>BasicServlet</servlet-name>
       <url-pattern>*.Basic</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
       <servlet-name>BasicServlet</servlet-name>
       <url-pattern>/BasicServlet</url-pattern>
    </servlet-mapping>
```

Three short lines of code versus 12 lines of XML elements and subelements. Are you starting to feel the power (and joy) of XDoclet? XML was meant to be parsed by computers and readable by humans, but not written by humans necessarily.

Step five: define J2EE resources

In addition to the above you can define resources reference in the *web.xml* for resources like the JDBC data sources or even ejb references. The Java file includes these XDoclet JavaDoc style tags at the class level as follows:

```
/** ...
 * @web.resource-ref description="JDBC resource"
 *                    name="jdbc/mydb"
 *                    type="javax.sql.DataSource"
 *                    auth="Container"
...
*/
public class BasicServlet extends HttpServlet {
```

The above generate the following elements in the *web.xml* file:

```
...

    <resource-ref>
        <description>JDBC resource</description>
        <res-ref-name>jdbc/mydb</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
    </resource-ref>
```

Using the **webdoclet** task to generate Web application deployment descriptors

This is all fine and dandy, but how do you take the Java source files with the fancy JavaDoc like tags and generate the *web.xml* file? For this, you need to write an Ant build file that uses the **webdoclet** task from XDoclet.

The **webdoclet** task is an Ant task for generating all manner of support files. For this example you will use the **deploymentdescriptor** subtask. But before you do this you have to setup XDoclet to be accessible from your Ant scripts. While you are setting up XDoclet, you might as well setup your environment.

Before you can run the **webdoclet** task you need to setup your environment with XDoclet and Ant. This will require installing and configuring XDoclet. And, then letting your Ant build file know about your application server configuration.

To do all of this you will need to:

1. Install XDoclet
2. Download and install the example
3. Configure the example to find your XDoclet install
4. Configure the example to find your application server's live deploy directory
5. Configure the example to find your application server's lib directory

---

Installing XDoclet

Download XDoclet 1.2 or higher. *Go to the files section of the XDoclet project on sourceforge.* Look for the file that looks like *xdoclet-bin-1.2xxx.zip* . Click on it.

Unzip the zip file in to a subdirectory called *xdoclet* off of your root directory. You can unzip in another place if you like.

I've included an example *build.xml* file that imports a properties file. The properties file sets all of the properties for you with few external references. If you did not install XDoclet in the root, which is very likely if you are using Unix, you will need to set the `xdocletlib` property in the *build.properties* file (More on this in the next slide).

Install and configure the example

You can download the example as a zip file. Unzip this file in your root directory. It will create a folder called *tutorials*.

If you are using the example, and you installed XDoclet in another location then all you have to do is modify the `xdocletlib` variable in the *build.properties* file (*tutorials\J2EEXdoclet\webdoclet\build.properites*).

Here is the listing of the *build.properties* file:

```
############### Change These for your environment #################

    # This is where you installed xdoclet
xdocletlib=/xdoclet/lib

    # Change these for your app servers.
    # This is the deployment directory.
webapps=/tomcat4/webapps
    # This lib is where the ant script expects to find the j2ee jar files.
lib=/tomcat4/common/lib
```

```
# You should not have to change these
src=./src
WEBINF=./web/WEB-INF
dest=${WEBINF}/classes

docroot=./web
output=./tmp

appname=webdoclet

# You may change these at will. These get used by the Servlet example.
basic.servlet.bye=dude
basic.servlet.hi=mom
```

The instructions in the *build.properties* file should be enough for the veteran Ant/J2EE developer to continue. For those who are Ant neophytes or are otherwise confused, read the next page carefully.

Details about Configuring build.properites for your environment

You need to change three setting in the *build.properties* file (*tutorials\J2EEXdoclet\webdoclet\build.properites*) as follows:
1. Set the location of the XDoclet install.
2. Set the hot deploy directory of your J2EE application server
3. Set the lib directory of your J2EE application server

Set the XDoclet lib

The default setting of `xdocletlib` is */xdoclet/lib*. If you have installed XDoclet in another location, please adjust this setting accordingly.

Set the deploy directory

You need to adjust the `webapps` property of the *build.properties* file to match your J2EE application server's hot deploy directory.

For example on my box: `webapps=/resin/webapps` would deploy to **Resin** while `webapps=/tomcat4/webapps` would deploy to **Tomcat 4**. The Ant build file will copy a war to whatever location you specify with the `webapps` property when you run the `deploy` target. **Set the J2EE lib directory**

The next thing you need to do is specify the lib directory where the Ant build file will find the J2EE jar files. You can do this by modifying the lib property of the *build.properties* file.

For example on my box: `lib=/resin/lib` would use Resin EE lib directory, `lib=/tomcat4/common/lib` would use the Tomcat lib directory and

`lib=/j2sdkee1.3.1/lib` would use Sun's reference implementation directory. The Ant build file uses this property to set the J2EE library jar files on the classpath for compilation.

If all of this talk about Ant is just not making sense, that is, you are an Ant neophyte, then I have the perfect thing for you. Please read this sample chapter from *Mastering Tomcat on Developing Web Components with Ant* (by yours truly) at the following link: *"http://www.rickhightower.com/AntPrimer.pdf"*. It is an excellent primer on using Ant.

Defining the webdoclet task

In order to run the **webdoclet** task, you must define the XDoclet **webdoclet** task with as taskdef in your ant build file as follows:

```
<taskdef name="webdoclet"
        classname="xdoclet.modules.web.WebDocletTask"
        classpathref="xdocpath"
/>
```

You have to do this because XDoclet is not a built-in Ant task. Notice the class name of the Ant task handler is `xdoclet.modules.web.WebDocletTask`. Also notice that you reference a predefined classpath called `xdocpath`. The `xdocpath` was defined earlier and it uses the `xdocletlib` property that was setup earlier as follows:

```
<path id="cpath">
    <fileset dir="${lib}"/>
</path>

<path id="xdocpath">
    <path refid="cpath"/>
    <fileset dir="${xdocletlib}">
        <include name="*.jar"/>
    </fileset>
</path>
```

The taskdef must have the XDoclet jar files in its classpath as well as the J2EE lib directories. Once you have defined the **webdoclet** taskdef, you can use the **webdoclet** task.

You can find these tasks and tags in the ant build file called build.xml.

Using the webdoclet task

To generate the *web.xml* file from the Java source, you need to use XDoclet's **webdoclet** task, as shown:

```
<webdoclet destdir="${dest}">

    <fileset dir="${src}">
        <include name="**/*Servlet.java" />
    </fileset>

    <deploymentdescriptor servletspec="2.3"
                          destdir="${WEBINF}" />

</webdoclet>
```

The **webdoclet** task is used in a Ant target called `generateDD`. The output directory of is specified with the **webdoclet** task's `destdir` attribute `webdoclet destdir="${dest}"`. The dest property is set in the *build.properties* file (/tutorials/J2EEXdoclet/webdoclet/web/WEB-INF).

The input files for the **webdoclet** task are specified with a fileset: `fileset dir="${src}"`. The src property is set in the *build.properties* file (*/tutorials/webdoclet/src*). The fileset uses a filter so that only the classes ending in Servlet will be selected as in BasicServlet: `<include name="**/*Servlet.java"/>`. This is to avoid processing all of the source code instead of just the source code that uses XDoclet tags.

The **deploymentdescriptor** subtask specifies the location for the generated deployment descriptor using the destdir attribute: `deploymentdescriptor ... destdir="${WEBINF}"`. The **deploymentdescriptor** is the subtask that actually generates the *web.xml* file.

Running Ant

To run the sample Ant build file, go to the directory that contains the project files. To run Ant, navigate to the */tutorials/J2EEXdoclet/webdoclet* directory and type: **ant deploy**

As I stated earlier, Ant will locate *build.xml*, the default name for the build file. (You may have to adjust your *build.properties* files.) For this example, here is the command-line output you should expect:

```
C:\tutorials\J2EEXdoclet\webdoclet>ant deploy
Buildfile: build.xml

init:
    [mkdir] Created dir: C:\tutorials\J2EEXdoclet\webdoclet\tmp\war

compile:
    [javac] Compiling 2 source files to
    C:\tutorials\J2EEXdoclet\webdoclet\web\WEB-INF\classes

generateDD:
[webdoclet] Running <deploymentdescriptor/>
[webdoclet] Generating web.xml.
```
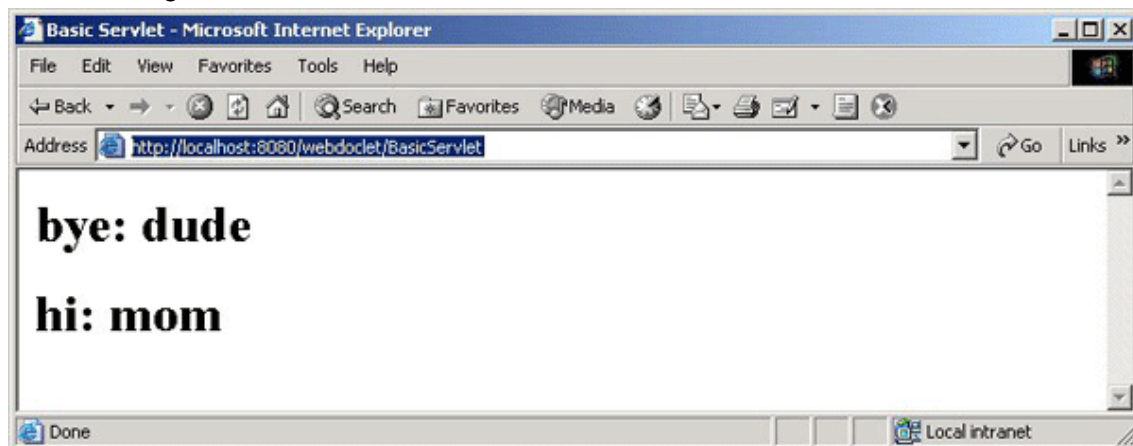
```
package:
      [war] Building war: C:\tutorials\J2EEXdoclet\webdoclet\tmp\war\webdoclet.war

deploy:
      [copy] Copying 1 file to C:\tomcat4\webapps

BUILD SUCCESSFUL
Total time: 13 seconds
```

Now that you have deployed it let's test it. Go to
http://localhost:8080/webdoclet/BasicServlet. You may have to adjust the port number
and/or context depending on your app server.

You should get a browser that looks like this:

Now open up the *build.properties* file and change the following properties like so:

```
basic.servlet.hi=I love XDoclet
basic.servlet.bye=Feel the power of XDoclet
```

Next, if you have an IDE like Eclipse that supports refactoring, change the package
name of the Servlet to com.foobar.ibm. Now rerun the ant build file as before (run the
clean target first then the deploy), and rerun the application. You see: The *web.xml* file
is in sync with the new changes. See the power. You are free to change the name or
package of the class as needed. XDoclet will keep the *web.xml* file in sync. Feel the
power of XDoclet!

Running Ant with Eclipse

If you are using Eclipse, you never have to leave the IDE to run Ant and your
application server (there are many plug-ins for many application servers).
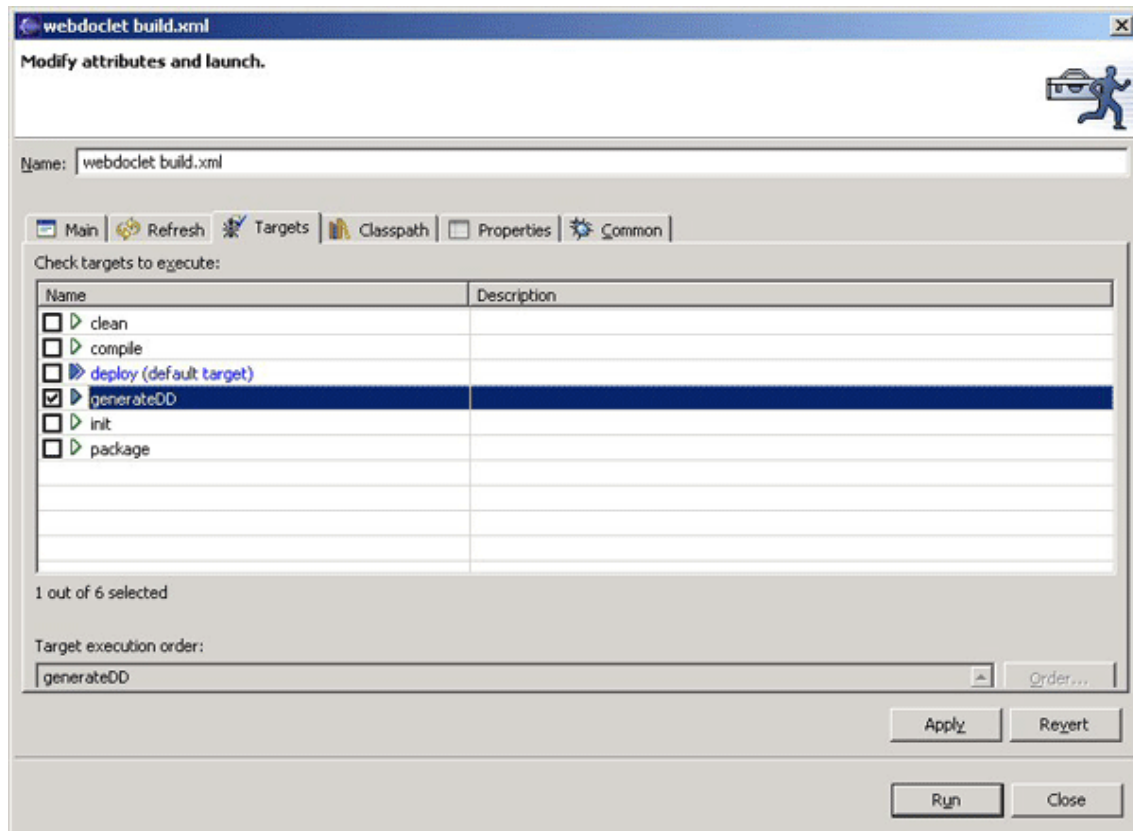
If you don't run the Ant build file inside of Eclipse when you are working with EJB technology you have to sync up Eclipse by running refresh and the rebuilding the project; however, if you run the Ant build file inside of Eclipse it automatically refreshes.
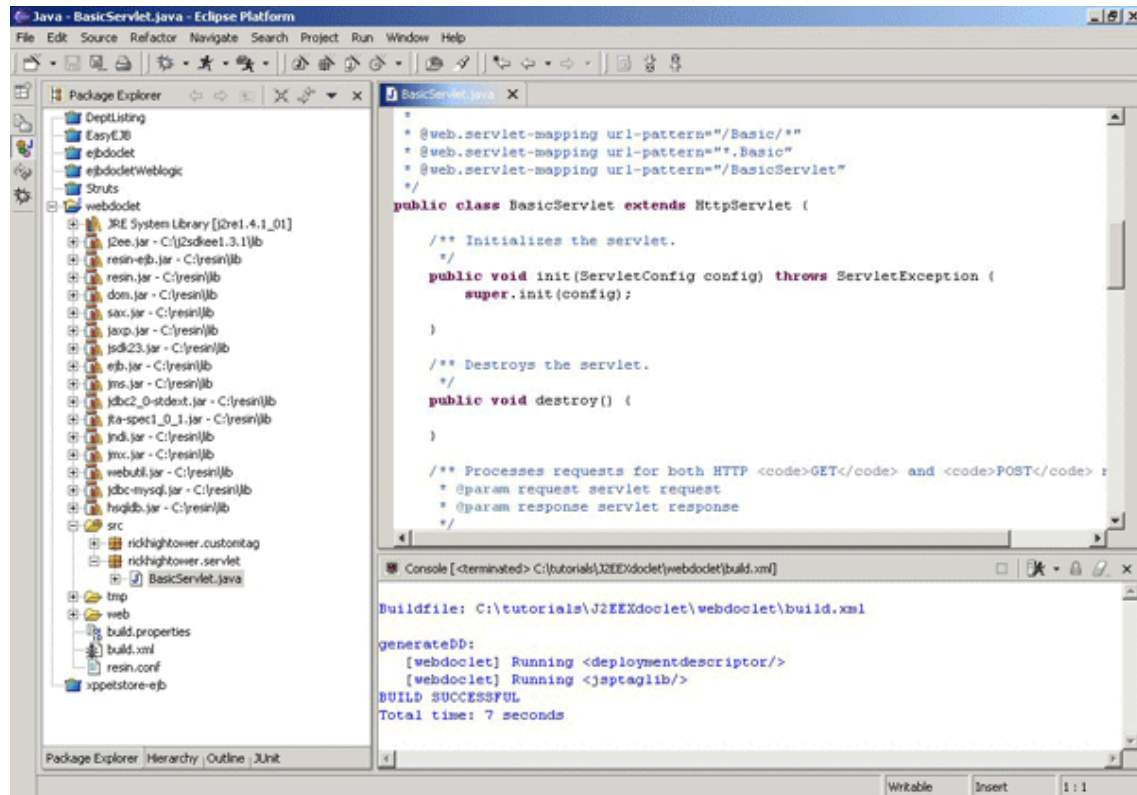
Section 3. Step-by-step Custom Tag example (TagLib)

Using the XDoclet?s webdoclet task to create Custom Tag TLDsClass level JavaDoc tags for Custom Tags

Just like before, the mappings are quite natural. The first step is to define the jsp tag using the jsp.tag and passing the name of the custom tag as follows:

```
 * @jsp.tag name="BasicTag"
```

This code generates the following in the TLD file:

```
    <tag>
        <name>BasicTag</name>
        <tag-class>tomcatbook.customtag.BasicTag</tag-class>
...
    </tag>
```

Remember that XDoclet uses the JavaDoc API to get the full class name of the custom tag handler. Next, you define any variables that you want available to the JSP pages that use your custom tag. For this example, you defined three variables. One of the variables can be used after the begin tag, one after the end tag only, and one only inside the body, as shown here:

```
 * @jsp.variable name-given="currentIter"
 *               class="java.lang.Integer" scope="NESTED"
 * @jsp.variable name-given="atBegin"
 *               class="java.lang.Integer" scope="AT_BEGIN"
 * @jsp.variable name-given="atEnd"
 *               class="java.lang.Integer" scope="AT_END"
```

A side benefit of XDoclet it that it keeps everything that makes of this Custom tag together in one file. In addition, it is great for documenting what makes up this Custom tag. Imagine not using XDoclet, you would have to go spelunking through a long TLD file looking for the right entries (the *struts-html.tld* file is 3000 lines long!) to see what variables this tag defined.

This code generates the following in the TLD file within the basic tag definition:

```
        </p><p>
         <variable>
                 <name-given>currentIter</name-given>
                 <variable-class>java.lang.Integer</variable-class>
                 <scope>NESTED</scope>
         </variable>
         <variable>
                 <name-given>atBegin</name-given>
```

```
                    <variable-class>java.lang.Integer</variable-class>
                    <scope>AT_BEGIN</scope>
            </variable>
            <variable>
                    <name-given>atEnd</name-given>
                    <variable-class>java.lang.Integer</variable-class>
                    <scope>AT_END</scope>
            </variable>
```

Method level JavaDoc tags for Custom Tags

Now here is a little something different. In the servlet example, all of the special
JavaDoc tags were at the class level. The custom tag example uses JavaDoc tags at
the method level to define custom tag attributes for the three attributes in this
example--includeBody, includePage, and iterate:

```
    /** Getter for property includePage.
     * @return Value of property includePage.
     * @jsp.attribute   required="true"
     *                  rtexprvalue="true"
     *                  description="The includePage attribute"
     */
    public boolean isIncludePage() {
        return this.includePage;
    }

 ...

    /** Getter for property includeBody.
     * @return Value of property includeBody.
     * @jsp:attribute   required="true"
     *                  rtexprvalue="true"
     *                  description="The includeBody attribute"
     */
    public boolean isIncludeBody() {
        return this.includeBody;
    }
 ...
    /** Getter for property iterate.
     * @return Value of property iterate.
     * @jsp:attribute   required="true"
     *                  rtexprvalue="true"
     *                  description="The iterate attribute"
     */
    public int getIterate() {
        return this.iterate;
    }
```

Note that the JavaDoc tag jsp.attribute is used to define the property as an
attribute. This code generates the following in the TLD file within the definition:

Custom tags, with all of their variables and attributes, can be hard to manage and keep
in sync. As you can see from the example, XDoclet can make short order of what

would otherwise be chaos, and it allows you to define all of this needed metadata in one file instead of two. This makes doing custom tags a lot easier--maybe easy enough for you to start fitting Custom Tags into your project.

Imagine refactoring and changing the name of a getter and setter method corresponding to an attribute. Without XDoclet you would have to spelunk through the TLD file. What a pain!

This is all well and good, but how do you take the Java source files and generate the TLD file?

Adding TagLib generation to XDoclet

You need to write an Ant script that uses the **webdoclet** task from XDoclet. The listing below modifies the code from the earlier **webdoclet** listing to add support for custom tags under the target `generateDD`. Just as before, the input files for the **webdoclet** task are specified with a nested fileset, except, this time you added a new include directive to include your tag handler (that is, `<include name="**/*Tag.java" />`). The jsptaglib subtask generates the TLD file as follows:

```
<webdoclet destdir="${dest}">

    <fileset dir="${src}">
        <include name="**/*Servlet.java" />
        <include name="**/*Tag.java" />
    </fileset>

    <deploymentdescriptor servletspec="2.3"
                          destdir="${WEBINF}" >
         <taglib uri="mytaglib"
                location="WEB-INF/tlds/mytaglib.tld"
          />

    </deploymentdescriptor>

    <jsptaglib
           jspversion="1.2"
           destdir="${WEBINF}/tlds"
           shortname="basic"
           filename="mytaglib.tld"/>

</webdoclet>
```

Notice that you added the subtask jsptaglib to create a TLD file called *mytaglib* under the `WEB-INF` directory of your Web application as follows:

```
<jsptaglib      jspversion="1.2"
                destdir="${WEBINF}/tlds"
                shortname="basic"
                filename="mytaglib.tld"/>
```

Also notice that you added a sub element under the **deploymentdescriptor** sub task to generate the taglib declaration in the *web.xml* file as follows:

```
<deploymentdescriptor servletspec="2.3"
                      destdir="${WEBINF}" >
     <taglib uri="mytaglib"
             location="WEB-INF/tlds/mytaglib.tld"
     />
```

The above would generate the following entry in the *web.xml* file as follows:

```
<taglib>
    <taglib-uri>mytaglib</taglib-uri>
    <taglib-location>WEB-INF/tlds/mytaglib.tld</taglib-location>
</taglib>
```

Testing your new JSP Custom Tag

Granted this JSP Custom tag is basic and does not do much--but it does work and demonstrates a lot of the features you can implement with Custom Tags There is a JSP file called *happy.jsp* in the docroot of this project. Once you run the Ant deploy target, you can edit the JSP file and try all the alternatives. Essentially, it will iterate the body as many times as you specify with the iterate attribute. The includeBody attribute flags specifies whether or not the body should be included, and the includePage attribute specifies whether or not the rest of the JSP file should be evaluated. I tried many permutations and it works as advertised.

```
<%@page contentType="text/html"%>
<%@taglib uri="mytaglib" prefix="mytag"%>
<html>
<head><title>I am a happy JSP page. Yeah!</title></head>
<body>

<mytag:BasicTag includePage="true" includeBody="true" iterate="2">
    Current iteration is <%=currentIter%> <br />
</mytag:BasicTag>

</body>
</html>
```

Section 4. Step-by-step EJB technology example

XDoclet and EJB technology

I started using XDoclet because of EJB technology. It allowed me to port my EJB components to many J2EE application servers. This is especially important with CMP as XDoclet generates vendor specific mappings for CMP/CMR to RDBMS. I had my own EJB code generator written in Jython but my cohort, Erik Hatcher, kept telling me how great XDoclet was until I finally broke down and tried it.

XDoclet does more than just facilitate porting of EJB components, it also makes EJB development much easier. How so? Now instead of having 5 or more files for a single EJB component, you only have one source file to work with. This will make an XDoclet fan out of any EJB developer.

Think about it. Now instead of maintaining a primary key class, local, and remote interfaces, local and remote homes, value classes, deployment descriptors, multiple vendor specific deployment descriptors, and more; I just have one file to maintain. I just use JavaDoc tags to markup my implementation class and XDoclet takes care of the rest. This is more than cool. This makes EJB technology easier to use. This helps fulfill the vision of component architecture.

Some vendors have started supporting XDoclet as part of their tools that ship with the J2EE application server (JRun). Hopefully this will be a trend. I'd love to see IBM and BEA ship their products with XDoclet support. Currently the vendor specific XDoclet templates lag behind the release a few months. Don't bother asking for them sooner. The XDoclet developers will tell you to implement them yourselves. XDoclet is open source in case you forgot. Actually the templates are pretty easy to modify, perhaps in a follow-up tutorial, I will add CMP/CMR support to some vendors product.

Note if you are an EJB neophyte, there are plenty of resources listed in the resource section to help you understand this section better. However, this section does assume some prior experience with EJB technology.

XDoclet is nice for Servlets, really nice for Custom Tags and just about essential for EJB technology. I'll cover how to use XDoclet with EJB with another step-by-step example.

XDoclet and EJB technology continued

Let's define two CMP entity beans (Dept and Employee) that have a one-to-many relationship (Dept has many Employees) and a Session bean (HRSystem) that can
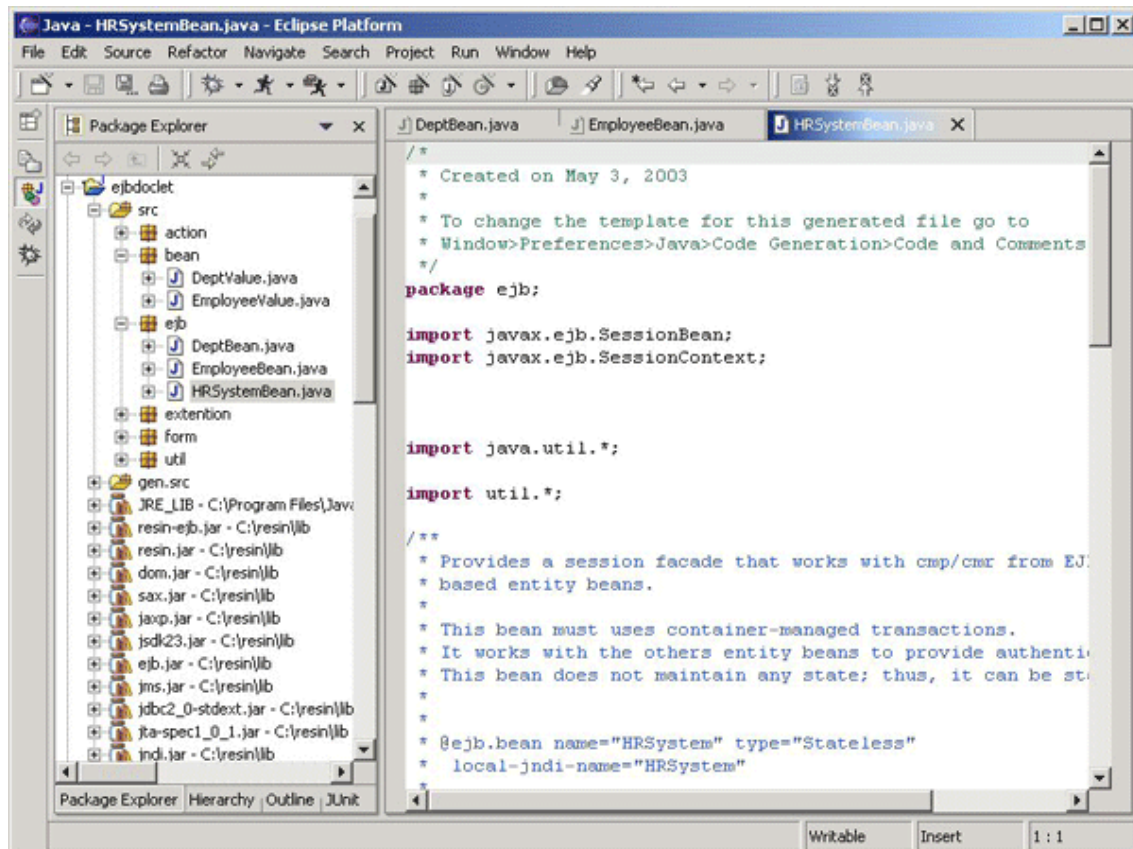
Here are the steps to complete this section:

1. Use `ejb.bean` tag to declare the EJB component's structure
2. Declare the names of the home and remote interfaces to be generated with ejb.home and `ejb.interface` tags.
3. Specify class level Object Relation (OR) Mapping: Map the Entity to a table with vendor specific tags
4. Specify finder methods for EntityBeans with `ejb.finder` tag
5. Mark the create method with `ejb.create` tag
6. Mark the primary key cmp field with `ejb.pk-field`
7. Mark the fields that are persistent
8. Set the vendor specific OR mapping for cmp fields
9. Add methods to an interface with `ejb.interface`
10. Setup a relationship between two Entities with `ejb.relation` tag
11. Setup OR relationship mappings between two Entities
12. Working with Session beans
13. Adding EJB references from the Session to the Entity with `ejb.ejb-ref` tag
14. Using the XDoclet Ant tag `ejbdoclet` to generate EJB support files

XDoclet and EJB: ejb.bean

The `ejb.bean` allows you to specify the type of the bean. The first bean I will cover will be an CMP 2.0 entity bean. You will need to specify that it is a CMP bean, that is uses `cmp-version` 2.x, give it a schema name, and specify its primary key (if it is not using a complex primary key), and the primary key type. Here is the Xdoclet tags for ejbs as demonstrated with the Dept bean:

```
/**...
 * @ejb.bean
 *     type="CMP"
 *     cmp-version="2.x"
 *     name="DeptBean"
 *     schema="Dept"
 *     local-jndi-name="DeptBean"
 *     view-type="local"
 *     primkey-field="id"
 *
 *
 * @ejb.pk               class="java.lang.Integer"
 *
    */
 public abstract class DeptBean implements EntityBean {
```

Notice that the primary key class is specified with the tags class attribute. The names are very close to what you would expect them to be. It is fairly intuitive as the corresponding meta-data matches that which you would find for the *ejb-jar.xml* file. What may not be obvious is the `view-type` parameter. The `view-type` parameter specifies if this is going to be a local bean, a remote bean or both. The above XDoclet tags would generate the following elements in the EJB deployment descriptor (*ejb-jar.xml*):

```
        <entity >
           <description>This entity bean represents a Department of Employees.
           </description>

           <ejb-name>DeptBean</ejb-name>

         ...

           <ejb-class>ejb.DeptBean</ejb-class>
           <persistence-type>Container</persistence-type>
           <prim-key-class>java.lang.Integer</prim-key-class>
           <reentrant>False</reentrant>
           <cmp-version>2.x</cmp-version>
           <abstract-schema-name>Dept</abstract-schema-name>
           ...
           <primkey-field>id</primkey-field>
```

Notice that the `<primkey-field>` specifies the single cmp field that is used for the primary key. If you were using a compound primary key then you would need to create a primary key class. XDoclet can easily generate the primary key class for you.

Another important point to notice is that you never specified the class of the EJB component directly. XDoclet, which parses the Java code, has access to the parse tree thus it knows the package name and class name already. This also means that if you change the name or package name, you would not have to manually update the deployment descriptor.

XDoclet and EJB technology: ejb.home and ejb.interface tags

XDoclet will generate the homes and interfaces for the class. You can give the interfaces that will be generated a name using `ejb.home` and `ejb.interface` tags. Or you can let XDoclet pick a name based on a pattern that you specified. In this example, you specify the name of the local home and local interface in the class file.

```
/**...
 * @ejb.home generate="local" local-class="ejb.DeptHome"
 * @ejb.interface generate="local" local-class="ejb.Dept"
 *  ...
 */
public abstract class DeptBean implements EntityBean {
```

Notice that with the `ejb.home` and `ejb.interface` you can specify to generate the local, remote or both. The above will cause an `ejb.DeptHome` home interface to be generated and a local interface called `ejb.Dept` to be generated. In addition the following elements will be defined in the *ejb-jar.xml* deployment descriptor based on the above tags.

```
<entity >
   ...

   <ejb-name>DeptBean</ejb-name>

   <local-home>ejb.DeptHome</local-home>
   <local>ejb.Dept</local>
   ...
```

XDoclet and EJB technology: ejb.persistence tag

There is no standard *OR* mapping defined by the EJB specification. But most application servers that implement EJB CMP CMR use a similar strategy, that is, mapping Entities to classes and fields to columns. The DeptBean EJB component is going to be mapped to a table called *TBL_USER*.

In older versions of XDoclet, every vendor implementation had their own support for EJB CMP CMR, yet they all seemed to pick different tags names for doing the same thing. The newer version of XDoclet defined a tag called `ejb.persistence` to specify the table name that the entity maps to.

I noticed the Resin XDoclet templates did not support the new `ejb.persistence` tag yet. I added the support but the new template was not committed into the project yet. It was actually really easy to modify the template to add the support for the new tag.

Here is the code to map the database table to the entity using the vendor specific Resin mapping, and the vendor neutral way. Currently not all vendor products are supported by the new tags. The vendor specific mapping tag is not needed if the templates for the vendor were updated to support ejb.persistence.

```
/**...
 * @resin-ejb.entity-bean     sql-table="DEPT"
 * @ejb.persistence           table-name="TBL_USER"
 *   ...
 */
public abstract class DeptBean implements EntityBean {
```

The above tags generates the following in the vendor specific mapping files. Resin is shown as an example.

```
    <!-- generated from ejb.DeptBean -->
        <entity>
            <ejb-name>DeptBean</ejb-name>
            <sql-table>DEPT</sql-table>

            ...
```

Notice that this example only works with these one application servers, but nothing stops you from using many others. Currently, XDoclet supports Orion, Pramati, IBM WebSphere, BEA WebLogic, JRun, JBoss, and a few more.

Not every implementation has the best support and the most up-to-date templates. It is open source so support will vary. However, it is fairly easy to update existing templates to work with other vendors.

Macromedia's JRun supports the templates themselves to ease development, that is, the commercial company supports and updates the XDoclet templates and modules. I hope this becomes a trend. However, it is fairly easy to modify the templates and create modules, and there are a lot of examples to look at as a basis.

XDoclet and EJB technology: ejb.finder

To define the `finder` method for the Entity bean, you just add the `ejb.finder` tag at the class level to identify the `finder` method signature with its EJB-QL query as follows:

```
/**
 *
 * @ejb.finder
 *    signature="Dept findByDeptName(java.lang.String name)"
 *    unchecked="true"
 *    query="SELECT OBJECT(dept) FROM Dept dept where dept.name = ?1"
 *    result-type-mapping="Local"
 *
 * @ejb.finder
 *    signature="Collection findAll()"
 *    unchecked="true"
 *    query="SELECT OBJECT(dept) FROM Dept dept"
 *    result-type-mapping="Local"
 *
 */
public abstract class DeptBean implements EntityBean {
```

This will define two `finder` methods in the generated home as well as `<query>` element definitions in the EJB deployment descriptor as follows:

```
/*
 * Generated by XDoclet - Do not edit!
 */
package ejb;

/**
 * Local home interface for DeptBean.
 */
public interface DeptHome
    extends javax.ejb.EJBLocalHome
{
    ...

    public ejb.Dept findByDeptName(java.lang.String name)
        throws javax.ejb.FinderException;

    public java.util.Collection findAll()
        throws javax.ejb.FinderException;

    public ejb.Dept findByPrimaryKey(java.lang.Integer pk)
        throws javax.ejb.FinderException;

}
```

```
        <entity >
           <description>This entity bean represents a Department of Employees.</descript
           <ejb-name>DeptBean</ejb-name>

           <local-home>ejb.DeptHome</local-home>
           <local>ejb.Dept</local>

            ...

           <query>
              <query-method>
                 <method-name>findByDeptName</method-name>
                 <method-params>
                     <method-param>java.lang.String</method-param>
                 </method-params>
              </query-method>
              <result-type-mapping>Local</result-type-mapping>
              <ejb-ql>SELECT OBJECT(dept) FROM Dept dept where dept.name =?1
              </ejb-ql>
           </query>
           <query>
              <query-method>
                 <method-name>findAll</method-name>
                 <method-params>
                 </method-params>
              </query-method>
              <result-type-mapping>Local</result-type-mapping>
              <ejb-ql>SELECT OBJECT(dept) FROM Dept dept</ejb-ql>
           </query>
         <!-- Write a file named ejb-finders-DeptBean.xml if you
         want to define extra finders. -->
         </entity>
```

XDoclet and EJB: ejb.create tag

The ejb.create tag is used to identify create methods. All create methods
(ejbCreateXXX) will have corresponding create methods generated in the generated
home by XDoclet. Here is an example of using the ejb.create tag.

```
 public abstract class DeptBean implements EntityBean {

    /**
     *
     * @ejb.create-method
     */
    public Integer ejbCreate(String name)
                                               throws CreateException {
        setName(name);

        return null;
    }
```

The above `ejb.create` tag would cause XDoclet to generate the following create method in the home.

```
 /*
  * Generated by XDoclet - Do not edit!
  */
package ejb;

 /**
  * Local home interface for DeptBean.
  */
public interface DeptHome
    extends javax.ejb.EJBLocalHome
{
   ...
   public ejb.Dept create(java.lang.String name)
      throws javax.ejb.CreateException;

   ...

}
```

XDoclet and EJB technology: ejb.pk-field

The `ejb.pk-field` marks a CMP field as participating as part of a compound key. You do not need this for this example. But if the `DeptBean` had a complex you would need it.

```
    /**
     * This is a cmp field.
     * And it is the primary key.
     *
     * @ejb:pk-field
     */
   public abstract Integer getId();
   public abstract void setId(Integer id);
```

XDoclet and EJB technology: ejb.persistent-field

The ejb.persistent-field tag marks a getter method as being part of a CMP field

declaration. This will cause the corresponding cmp-field elements to be generated in the EJB deployment descriptor:

```
       /**
        * This is a cmp field.
        * And it is the primary key.
        *
        * @ejb.persistent-field
        ...
        */
   public abstract Integer getId();
   public abstract void setId(Integer id);
```

The above XDoclet tag would cause the following `<cmp-field>` to be generated.

```
     <entity >
        <description>This entity bean represents a Department of
                     Employees.</description>

        <ejb-name>DeptBean</ejb-name>

         ...
        <cmp-field >
           <description>This is a cmp field.</description>
           <field-name>id</field-name>
        </cmp-field>
```

Notice that with the ejb.persistent-field you do not specify the name of the field. Thus if you changed the names of the CMP field, that is, `getTheID`, `setTheID`, you would not have to manually sync the deployment descriptor. This is the beauty of XDoclet; it knows about the context of the declaration. These are the features that make refactoring code easier.

XDoclet and EJB technology: Set the vendor specific OR mapping with the vendor specific tag (or not)

The `ejb.persistence` tag allows you to specify the mapping from a cmp field to a table column. Here is an example of using the `ejb.persistence` tag with the `DeptBean`'s ID CMP field as follows:

```
       /**

        * This is a cmp field. The cmp field is read only.
        * And it is the primary key.
        *
        * @ejb.pk-field
        * @ejb.persistent-field
```

```
        * @ejb.interface-method view-type="local"
        * @ejb.persistence        column-name="DEPTID"
        * @resin-ejb.cmp-field    sql-column="DEPTID"
        */
    public abstract Integer getId();
    public abstract void setId(Integer id);
```

Notice that the above ejb.persistence maps the ID CMP field to the DEPTID column. (In this example I also use the vendor specific mapping tag since not all vendor templates support the new `ejb.persistence` tag yet). The above XDoclet tags would generate the following mappings in the vendor specific RDBMS mapping files.

Resin's resin.ejb

```
    <!-- generated from ejb.DeptBean -->
        <entity>
            <ejb-name>DeptBean</ejb-name>
            <sql-table>DEPT</sql-table>

            <cmp-field>
                <field-name>id</field-name>
                <sql-column>DEPTID</sql-column>
            </cmp-field>
```

XDoclet and EJB technology: ejb.interface tag

To declare that a method in the implementation should show up in the remote or local interface you use the `ejb.interface` tag as follows:

```
     /**
    ...
        * @ejb:interface-method view-type="local"
        ...
        */
    public abstract Integer getId();
    public abstract void setId(Integer id);
```

The `ejb.interface` tag specifies a `view-type` parameter. The view-type parameter tells XDoclet where it should generate the method: local, remote or both for the local interface, the remote interface of both interfaces. The above XDoclet tag would cause the following method to be generated in the local interface as follows:

```
 /*
  * Generated by XDoclet - Do not edit!
  */
```

```
package ejb;

/**
 * Local interface for DeptBean.
 */
public interface Dept
    extends javax.ejb.EJBLocalObject
{
...
    /**
     * This is a cmp field. The cmp field is read only. And it is the primary key.
     */
    public java.lang.Integer getId(  ) ;

...

}
```

Notice that only the getter method was exported to the local interface because only the getter method had the XDoclet `ejb.interface` tag associated with it.

XDoclet and EJB technology: ejb.relation tag

The `ejb.relation` tag allows you to define a CMR relationship. The tag is fairly straight forward as the attribute names match the corresponding element names really closely in the EJB deployment descriptor.

The following example sets up a one to many relationship from the `DeptBean` to the `EmployeeBean`.

```
...
public abstract class DeptBean implements EntityBean {

    ...


    /**
     * @return return employees in this department
     *
     * @ejb.interface-method view-type="local"
     *
     * @ejb.transaction type="Required"
     *
     * @ejb.relation
     *     name="EmployeesInADepartmentRelation"
     *     role-name="DeptHasEmployees"
     *     target-role-name="EmployeeInADept"
     *     target-cascade-delete="no"
     */
    public abstract Collection getEmployees();
        /** @ejb.interface-method view-type="local" */
    public abstract void setEmployees(Collection collection);
```

```
   ...
```

Notice that the relationship is given a name. Then you describe the DeptBeans role in the relationship. You could also specify that when an instance of this bean is deleted that all of its children in the relationship are also deleted with the `target-cascade-delete` attribute. Notice that you also specified the target-role-name. The attributes that start with target are really only needed for one way relationships (unidirectional). Since this example is bi-directional you could have left this out. The multiplicity of the relationship is based on the return type of the getter method in the cmr field. Since `getEmployees` returns a Collection, XDoclet infers that the multiplicity of `Employee` is many.

No matter how flat a pancake is, its always got two sides just like an CMR relationship. Here is the other side of the CMR relationship.

```
 public abstract class EmployeeBean implements EntityBean {

     ...

   /**
    * @return Return the group this user is in.
    *
    * @ejb.interface-method view-type="local"
    *
    * @ejb.transaction type="Required"
    *
    * @ejb.relation
    *     name="EmployeesInADeptartmentRelation"
    *     role-name="EmployeeInADept"
    *     target-role-name="DeptHasEmployees"
    *
    */
   public abstract Dept getDept();
   /** @ejb.interface-method view-type="local" */
   public abstract void setDept(Dept dept);
```

Again the target attribute is a little bit of extra information that is really only needed in the case on a unidirection relationship, but it is here for good measure. Notice that the Relationship name is the same as the relationship name on the Dept side of the relationship. This is what XDoclet uses to correlated the two members of the relationship.

The above ejb.relation tags would generate the following entries in the deployment descriptor.

```
   <!-- Relationships -->
   <relationships >
     <ejb-relation >
       <ejb-relation-name>EmployeesInADeptartmentRelation</ejb-relation-name>
```

```
        <ejb-relationship-role >
           <ejb-relationship-role-name>EmployeeInADept</ejb-relationship-role-name>
           <multiplicity>Many</multiplicity>
           <relationship-role-source >
              <ejb-name>EmployeeBean</ejb-name>
           </relationship-role-source>
           <cmr-field >
              <cmr-field-name>dept</cmr-field-name>
           </cmr-field>
        </ejb-relationship-role>

        <ejb-relationship-role >
           <ejb-relationship-role-name>DeptHasEmployees</ejb-relationship-role-name>
           <multiplicity>One</multiplicity>
           <relationship-role-source >
              <ejb-name>DeptBean</ejb-name>
           </relationship-role-source>
           <cmr-field >
              <cmr-field-name>employees</cmr-field-name>
              <cmr-field-type>java.util.Collection</cmr-field-type>
           </cmr-field>
        </ejb-relationship-role>

     </ejb-relation>
  </relationships>
```

With XDoclet you had to write one tag with two attributes for each side of the
relationship. Compare this to the 26 lines of XML you would have had to have written
without XDoclet. I don't see how anyone would want to do EJB development without
XDoclet.

XDoclet and EJB technology: Setup OR relationship mappings between two Entities

So far you have defined the relationship between two beans. This does not amount to a
hill of beans, unless you add relationship mapping details about the underlying
database tables. The generated vendor implementations need these mappings so it
knows how to implement the relationship.

Unfortunately there is not a common way with XDoclet to specify the OR relationship
mappings. Currently you have to learn each set of vendor specific tags. Below is an
example that uses both the vendor neutral tag and Resin's OR relationship mapping
tag.

```
    /**
     * @return Return the group this user is in.
     *
         ...
     *
```

```
    * @resin-ejb.relation    sql-column="DEPTID"
    *
    *
    */
    public abstract Dept getDept();
    /** @ejb:interface-method view-type="local" */
    public abstract void setDept(Dept dept);
```

The `resin-ejb.relation` tag has a single parameter (sql-column) that points the column involved in the relationship, that is, the foreign key. It assumes that the foreign key is pointing to the primary key of the other side of the relationship.

Some vendor specific mappings take two parameters the foreign key and the name of the column in the other table (Dept) that the foreign key points to. To put this in perspective, let's show the actual SQL DDL for these tables.

```
 CREATE TABLE DEPT (
     DEPTID        INT           IDENTITY    PRIMARY KEY,
     NAME          VARCHAR (80)
 );


 CREATE TABLE EMPLOYEE (
     EMPID         INT           IDENTITY     PRIMARY KEY,
     FNAME         VARCHAR (80),
     LNAME         VARCHAR (80),
     PHONE         VARCHAR (80),
     DEPTID        INT,
     CONSTRAINT  DEPTFK FOREIGN KEY (DEPTID)
                      REFERENCES DEPT (DEPTID)
 );
```

The employee table has a foreign key `DEPTID` that references the department's primary key `DEPTID`.

XDoclet and EJB technology: Working with Session beans

Now that you defined some Entity beans let's create a session bean that access the Entity beans as follows:

```
 /**
  * Provides a session facade that works with cmp/cmr from EJB 2.0
  * based entity beans.
  *
  * This bean must uses container-managed transactions.
  * It works with the others entity beans to provide authentication services.
  * This bean does not maintain any state; thus, it can be stateless.
```

```
 *
 *
 * @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 group names.
     *

     * This method is part of the ejb interface.
     * @ejb.interface-method view-type="local"
     * @ejb.transaction type="Required"
     */
    public String[] getDepts() {
        ArrayList deptList = new ArrayList(50);
        Collection collection = LocalFinderUtils.findAll("DeptBean");
        Iterator iterator = collection.iterator();
        while (iterator.hasNext()) {
                Dept group = (Dept) iterator.next();
                deptList.add(group.getName());
        }
        return (String[]) deptList.toArray(new String[deptList.size()]);
    }
```

Notice that the Session bean use the same `ejb.bean` tag, but instead of specifying CMP, it specifies that this is a Stateless bean as follows:

```
/**
 * 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"
 *
 ...
 *
 */
public class HRSystemBean implements SessionBean {
      ...
```

The above would cause XDoclet to generate the following listing.

```
    <enterprise-beans>

        <!-- Session Beans -->
        <session >
            <description>provides a session facade that works with
            cmp/cmr from EJB 2.0 based entity beans.</description>


            <ejb-name>HRSystem</ejb-name>

            <home>ejb.HRSystemHome</home>
            <remote>ejb.HRSystem</remote>
            <local-home>ejb.HRSystemLocalHome</local-home>
            <local>ejb.HRSystemLocal</local>
            <ejb-class>ejb.HRSystemBean</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>
            ...
```

```
  I am so glad XDoclet generates that and I don't have to write it out.
```

XDoclet and EJB technology: Adding EJB references from the Session to the Entity with ejb.ejb-ref tag

In order for your session bean to access the entity beans that you created, you will need to add a reference to the entity beans. You can do this with the `ejb.ejb-ref` tag as follows:

```
/**
 ...
 * @ejb.ejb-ref ejb-name="DeptBean" view-type="local"
 * @ejb.ejb-ref ejb-name="EmployeeBean" view-type="local"
 *
 */
public class HRSystemBean implements SessionBean {
```

Note that the two little lines of tag code generate the following ejb-local-ref tag listings.

```
    <enterprise-beans>

        <!-- Session Beans -->
        <session >
            ...
            <ejb-name>HRSystem</ejb-name>
```

```
        ...

      <ejb-local-ref >
          <ejb-ref-name>ejb/DeptBean</ejb-ref-name>
          <ejb-ref-type>Entity</ejb-ref-type>
          <local-home>ejb.DeptHome</local-home>
          <local>ejb.Dept</local>
          <ejb-link>DeptBean</ejb-link>
      </ejb-local-ref>
      <ejb-local-ref >
          <ejb-ref-name>ejb/EmployeeBean</ejb-ref-name>
          <ejb-ref-type>Entity</ejb-ref-type>
          <local-home>ejb.EmployeeHome</local-home>
          <local>ejb.Employee</local>
          <ejb-link>EmployeeBean</ejb-link>
      </ejb-local-ref>
    ...
```

Two lines of XDoclet tags corresponds to 15 lines of XML in the deployment descriptor,
and when or if you refactor the included classes the deployment descriptor stays in
sync. This is highly significant.

XDoclet and EJB technology: Using the XDoclet Ant
task ejbdoclet

Now that you define all of your beans, you need to modify your Ant build script to use
the XDoclet JavaDoc tags to generate the dependent files as follows:

```
<target name="ejbdoclet" >

    <taskdef
        name="ejbdoclet"
        classname="xdoclet.modules.ejb.EjbDocletTask"
        classpathref="xdocpath"
        />

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

    >

        <fileset dir="${src}">
            <include name="ejb/*Bean.java" />
        </fileset>

        <localinterface/>
        <localhomeinterface />
        <remoteinterface/>
        <homeinterface />

        <entitypk/>
```

```
        <deploymentdescriptor
                destdir="META-INF"
                destinationFile="ejb-jar.xml"
                validatexml="true" />

        <deploymentdescriptor
                destdir="${WEBINF}"
                destinationFile="cmp-xdoclet.ejb"
                validatexml="true" />

            <resin-ejb-xml destDir="${WEBINF}"/>


    </ejbdoclet>
```

The above ant script defines the `ejbdoclet` task with `taskdef` task. It then uses the
`ejbdoclet` task to generate all manners of support files. It uses the nested fileset to
select the source that XDoclet will use in this case all classes ending with Bean. It then
uses the following subtasks: `localinterface`, `localhomeinterface`,
`remoteinterface`, `homeinterface` and `entitypk` to generate the following local
interfaces, local homes, remote interface, remote homes and primary key classes
respectively (This example does not have primary key classes as the entities use
simple one field keys).

Section 5. xPetstore

This tutorial just scratches the surface

I have only scratched the surface of what you can do with XDoclet and Ant. EJB technology alone has a dizzying array of features and support. XDoclet provides you with powerful tools in your toolbox. XDoclet modules make Web development easier. Ant is powerful tool, and XDoclet is an example how Ant can be extended to simplify Web development. I provided a small glimpse of Ant. For some this is enough; for others, a more detailed description is in order.

See the resource section for other examples of using XDoclet.

This tutorial just scratches the surface of things you can do, the xPetstore example gives much more detail, but still does not cover it all.

---

xPetstore

xPetstore re-implements Sun's Microsystem PetStore using XDoclet. It is an excellent source for information on how to use XDoclet with great examples. The xPetstore demonstrates the use of Ant and XDoclet to build WODRA (Write Once, Deploy and Run Anywhere) J2EE applications. There are currently two version of xPetsore.

The first version of xPetstore is a pure J2EE version that uses EJB components (CMPCMR 2.0), JSP, Struts, and Sitemesh. The second version of xPetstore is a Servlet based solution that uses Velocity, WebWork, Sitemesh, POJO, and Hibernate. XDoclet provides custom handlers, templates, and Ant tasks to do for Struts, WebWork, and Hibernate what this tutorial showed XDoclet can do for EJB, Custom Tags, and Servlets. The xPetstore shows how to develop EJB (CMP CMR 2.0), Custom Tags (TagLib), JSP, Struts, WebWork, Servlet Filters, Hibernate with XDoclet. Check out the xPetstore before you start using XDoclet on your project.

xPestore has been deployed and tested on the following platforms:

Operating system:
- Linux
- Windows

Application Servers:
- JBoss 3.x
- WebLogic 7.x

Databases:
- Hypersonic SQL
- PostgreSQL

- SapDB
- MySQL
- Oracle
- MS SQL Server

xPetstore demonstrates the following uses for XDoclet: generate EJB 2.0 files, home and business interfaces (local and remotes) for EJBs, EJB deployment descriptors (*ejb-jar.xml*), vendor specific deployment descriptors, ejb design patterns, the use of J2EE 1.3 features like CMP 2.0 and CMR, generate Web deployment descriptors for: Servlets, Web Filters and JSP Taglibs.

In addition the example demonstrates the how to generate Struts deployment descriptors, Webwork deployment descriptor, how to use technologies like Velocity, how to use persistence layers like Hibernate, and the use of XDoclet merge points. All of the components are tested using the JUnitEE testing framework.

xPetstore is a great resource for learning how to use XDoclet modules.

Section 6. Summary

Summary

This tutorial showed J2EE developers how to use XDoclet to speed development by showing three step-by-step tutorials for using XDoclet.

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.

You learned how to use XDoclet with Servlets, Custom Tags, and EJB Session and Entity beans.

XDoclet generates these related support files by parsing your source files similar to the way the JavaDoc engine parses your source to create JavaDoc documentation. XDoclet, like JavaDoc, not only has access to these extra tags that you added, but also access to the structure of your source. XDoclet applies this hierarchy tree of data and context to templates. It uses all of this to generate what would otherwise be monotonous support files.

XDoclet speeds development by being less verbose than corresponding deployment descriptors, by keeping the source in sync with the deployment descriptors and support files which enable refactoring, and lastly by generating a metric ton of support files; in the case of EJB one source to five generated files is not uncommon.

Resources

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*

Special thanks to the XDoclet team for developing such a useful tool.

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.