Create multi-purpose Web content with XSLT

Repurpose content without altering data

Skill Level: Introductory

Nicholas Chase (nicholas@nicholaschase.com)
Author
Studio B

18 Mar 2003

This tutorial is for developers who want to create content that can be repurposed for a variety of presentations without affecting the original data. It explains how to create a Java servlet that determines the means by which the data is being viewed and uses XML and XSLT to provide the appropriate presentation. It also includes information on using the servlet as the basis for a Web service.

Section 1. Introduction

Should I take this tutorial?

This tutorial is for developers who want to create content that can be repurposed for a variety of presentations without affecting the original data. It explains how to create a Java servlet that determines the means by which the data is being viewed and uses XML and XSLT to provide the appropriate presentation. It also includes information on using the servlet as the basis for a Web service.

This tutorial uses Java to transform XML data, but the XSLT concepts are common to other languages as well. It assumes that you are familiar with XML and with XSL Transformations, but the actual transformations themselves are a minor part of the tutorial. An understanding of Java development is helpful, but not required.

What is this tutorial about?

As the Web grows, it is increasingly being pulled in opposite directions. On the desktop, Web pages and the browsers that display them are becoming more and more complex as companies strive for presentations that are eye-catching while still being functional. At the same time, Web content is increasingly being repurposed for devices such as pagers and mobile phones that, while more robust than they used to be, are still incapable of handling some of these more complex pages. In addition, the future will see more and more content accessed as Web services, which typically requires an entirely different structure.

This tutorial takes a series of headlines stored in an XML file and explains how to automatically choose the proper XSLT style sheet to display them in a traditional browser or a mobile browser, and to return them as the response for a Web service, both through a proxy created in WebSphere Studio V5 and directly.

The tutorial covers:

- The basic environment
- Creating a servlet
- Performing a transformation
- Selecting the appropriate style sheet
- Creating and accessing the Web service

Tools

You can gain a good understanding of the concepts behind this tutorial without actually executing the examples, but should you choose to follow along, the following tools should be installed and tested prior to starting the tutorial:

- Although not required, all of the necessary tools for creating XML and XSLT files and Java classes, as well as creating the Web service proxy, are available within IBM's WebSphere Studio V5. WebSphere Studio also includes a test server environment for running the servlet, so you can accomplish all of the development in this tutorial with this one tool.
- To see the mobile phone examples, you can download the Openwave SDK, which includes a simulator that works over HTTP. Download the SDK at http://developer.openwave.com/download/product_62.html. The examples use only HTTP, so the WAP extension is not necessary for this tutorial. (Phone screenshots provided by Openwave.)

If you choose not to use WebSphere Studio or another development environment, you can also complete the examples using:

- A text editor to create and XML and XSLT files.
- A Java development environment such as Sun's Java 2 Standard Edition version 1.4, available at http://java.sun.com/j2se/1.4/. Version 1.4 has built-in XML support.
- A servlet-capable Web server such as IBM's WebSphere Application Server, or Apache Tomcat, available at http://tomcat.apache.org/tomcat-4.1-doc/. (Even if you use WebSphere Studio to create and test the application, you'll need a server on which to deploy your production application.)

Section 2. Project overview

What we want to accomplish

One of the arguments for storing content in XML rather than directly as HTML is that you can repurpose material for different media. Often companies and Web site developers fail to use this capability because they don't know how easy it can be.

This tutorial takes a set of headlines that are stored in an XML file and transforms them, using XSLT, into a variety of presentations. XSLT was specifically designed for this purpose, providing an easy way to indicate different style sheets for a single XML file.

Ultimately, your goal is to have a single servlet that analyzes any requests that come in and decides which XSLT style sheet to use to transform the data.

The source file

Because of the nature of this tutorial, the XML file itself is fairly simple:

```
<?xml version="1.0" encoding="UTF-8"?>
<news>
 <story storyid="1">
   <headline>New trailers online</headline>
   <blurb>Trailers for this year's summer movies debuted on TV
          during the Super Bowl.</blurb>
```

```
    <permalink>
       http://www.vanguardreport.com/phpnuke/modules.php?
       name=News&amp;file=rssArticle&amp;sid=645
    </permalink>
  </story>
  <story storyid="2">
    <headline>NASA responds to UFO claims</headline>
    <blurb>NASA has posted a response to EuroSETI's claims that
           they have found proof of the existence of aliens,
           explaining how these potential UFOs can be "created"
           from their images.</blurb>
    <permalink>
       http://www.vanguardreport.com/phpnuke/modules.php?
       name=News&amp;file=rssArticle&amp;sid=644
    </permalink>
  </story>
  <story storyid="3">
    <headline>Judge rules comic book mutants non-human</headline>
    <blurb>In a customs lawsuit that made strange bedfellows, a
           comic book company won a court judgement that its
           mutant heroes are not human.</blurb>
    <permalink>
       http://www.vanguardreport.com/phpnuke/modules.php?
       name=News&amp;file=rssArticle&amp;sid=640
    </permalink>
  </story>
  <story storyid="4">
    <headline>EuroSETI to announce UFO proof</headline>
    <blurb>A group studying images normally used to search for
           comets claims to have found photographic evidence of
           UFOs.</blurb>
    <permalink>
       http://www.vanguardreport.com/phpnuke/modules.php?
       name=News&amp;file=rssArticle&amp;sid=639
    </permalink>
  </story>
</news>
```

Note that the URLs in the `permalink` tags were split onto multiple lines for viewing purposes only. In reality, the URLs are on a single line.

This file simply lists a number of stories, including their headlines, a simple description, and a location where the complete story can be found.

The basic output

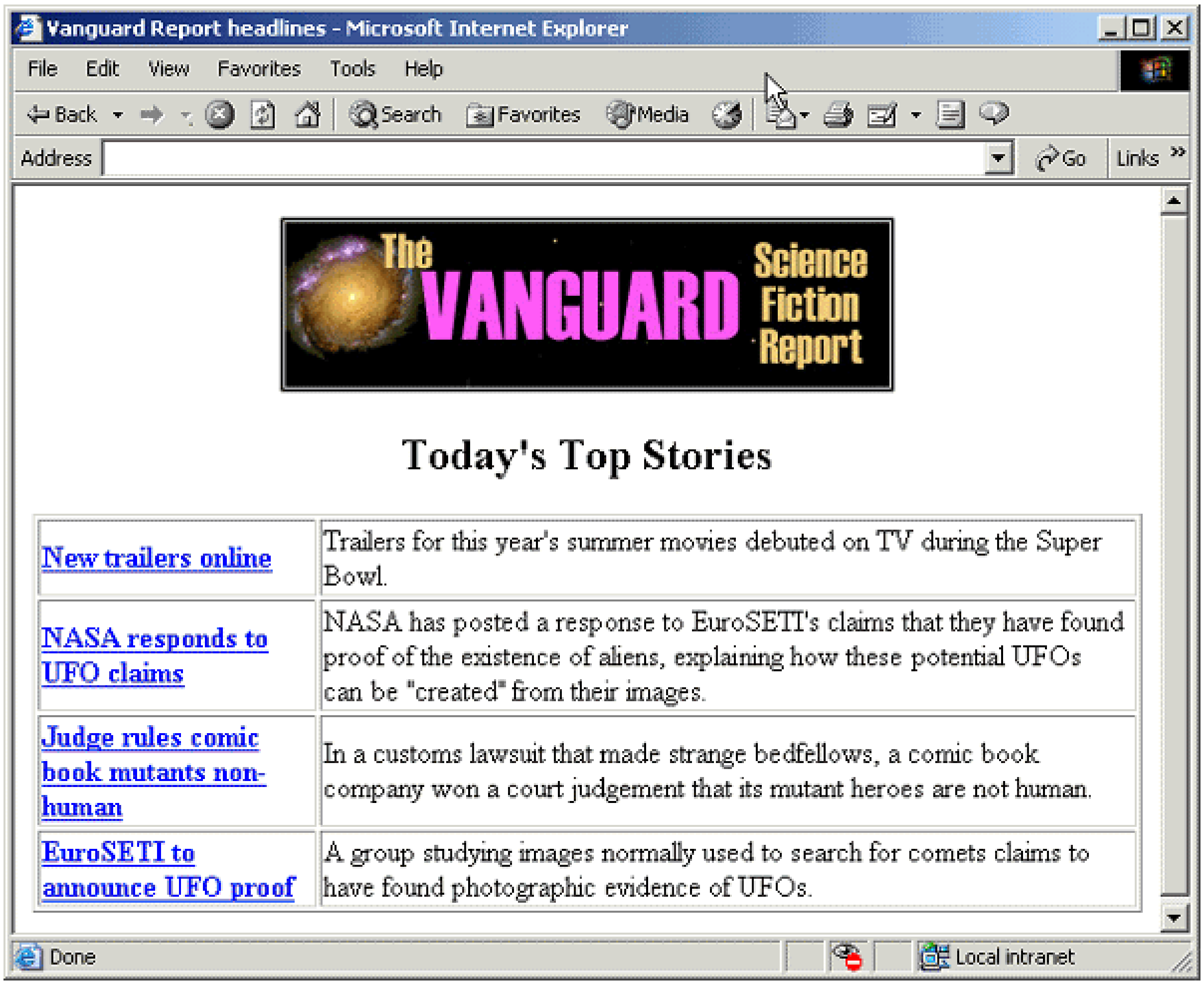
Given the lack of space in a tutorial (and the author's complete lack of visual design skills) the basic output page, intended for traditional desktop browsers, is fairly straightforward:

```
<?xml version="1.0" encoding="UTF-8"?>
<html>
<head><title>Vanguard Report headlines</title></head>
<body>
 <center>
  <img
src="http://www.vanguardreport.com/phpnuke/themes/3D-Fantasy/images/logo.gif"
alt="logo" width="340" height="100" />
 </center>
```

```
  <h2 align="center">Today's Top Stories</h2>

  <table>
  <tr><td><a href="/XSLProj/XSLServlet?storyid=1">
           <b>New trailers online</b></a>
       </td>
       <td>
         Trailers for this year's summer movies debuted on TV during
         the Super Bowl.
       </td></tr>
  <tr><td><a href="/XSLProj/XSLServlet?storyid=2">
           <b>NASA responds to UFO claims</b></a>
       </td>
       <td>NASA has posted a response to EuroSETI's claims that
         they have found proof of the existence of aliens, explaining
         how these potential UFOs can be "created" from their images.</td></tr>
  <tr><td><a href="/XSLProj/XSLServlet?storyid=3">
           <b>Judge rules comic book mutants non-human</b></a></td>
       <td>In a customs lawsuit that made strange bedfellows,
         a comic book company won a court judgement that its mutant heroes
         where not human.</td></tr>
  <tr><td><a href="/XSLProj/XSLServlet?storyid=4">
           <b>EuroSETI to announce UFO proof</b></a></td>
       <td>A group studying images normally used to search for comets
         claims to have found photographic evidence of UFOs.</td></tr>
  </table>
 </body>
 </html>
```

This is the page that would probably be accessed most frequently, appearing when the site is accessed with a traditional browser. When a headline is clicked, it goes to a page with more information on that particular story.

Figure 1. Basic output page for traditional desktop browsers


Vanguard Report headlines - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address Go Links
The VANGUARD Science Fiction Report
Done
Local intranet

Today's Top Stories

New trailers online	Trailers for this year's summer movies debuted on TV during the Super Bowl.
NASA responds to UFO claims	NASA has posted a response to EuroSETI's claims that they have found proof of the existence of aliens, explaining how these potential UFOs can be "created" from their images.
Judge rules comic book mutants non-human	In a customs lawsuit that made strange bedfellows, a comic book company won a court judgement that its mutant heroes are not human.
EuroSETI to announce UFO proof	A group studying images normally used to search for comets claims to have found photographic evidence of UFOs.

The mobile output

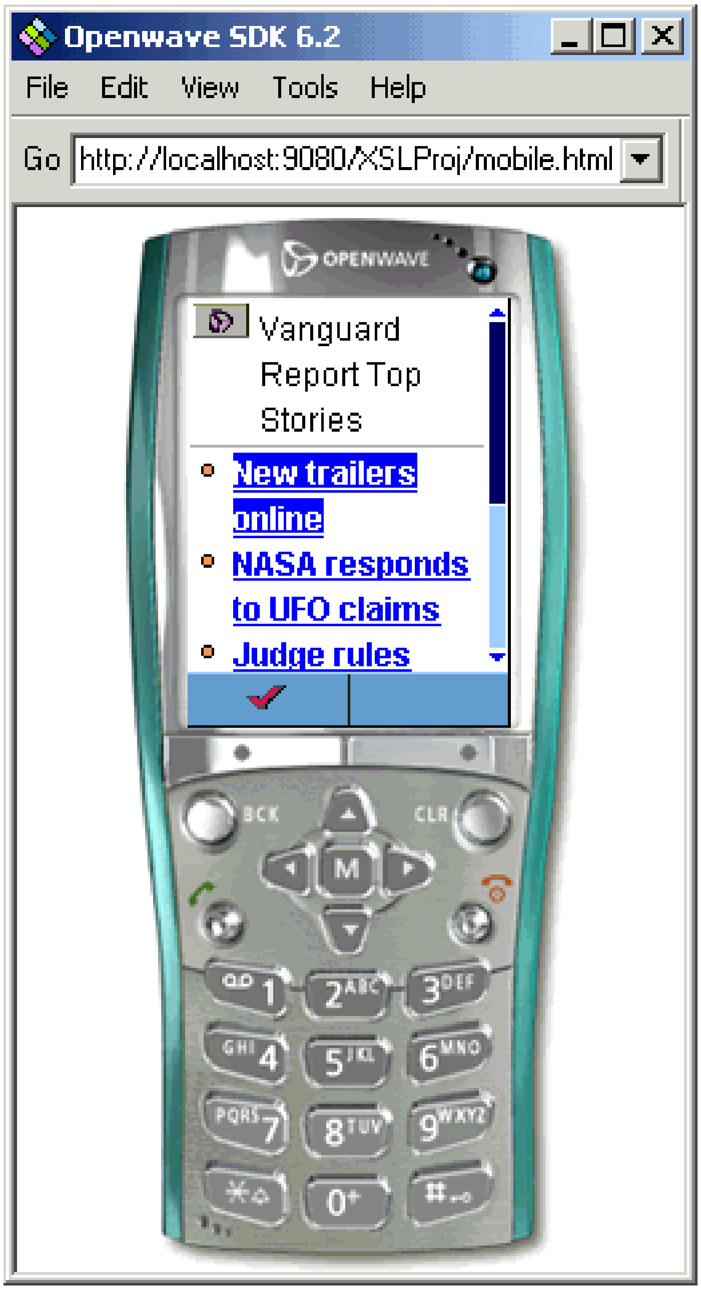
At one time, writing a page for a mobile phone involved learning an entirely new markup language. But the next generation of phones is equipped with a smaller version of a traditional browser, making it possible to create pages that are much more familiar to developers.

In this case, a simpler version of the original page, which uses a subset of XHTML known as XHTML Mobile, suffices:

```
<?xml version="1.0" encoding="UTF-8"?>
<html>
<head><title>Vanguard Report Top Stories</title></head>
<body>
<ul>
<li><a href="/XSLProj/XSLServlet?storyid=1">
             <b>New trailers online</b></a></li>
<li><a href="/XSLProj/XSLServlet?storyid=2">
             <b>NASA responds to UFO claims</b></a></li>
<li><a href="/XSLProj/XSLServlet?storyid=3">
             <b>Judge rules comic book mutants non-human</b></a></li>
```

```
<li><a href="/XSLProj/XSLServlet?storyid=4">
             <b>EuroSETI to announce UFO proof</b></a></li>
</ul>

</body>
</html>
```

Figure 2. Simpler version of original page created with XHTML Mobil


Openwave SDK 6.2
File Edit View Tools Help
Go http://localhost:9080/XSLProj/mobile.html
OPENWAVE
Vanguard Report Top Stories
New trailers online
NASA responds to UFO claims
Judge rules

The idea is that when one of these phones accesses the servlet, the servlet provides this simplified version.

Section 3. Setting up the environment

The Web environment

To make this scenario work, you're going to need a Web server that's capable of serving dynamic content. This tutorial uses Java servlets as its development platform, so ultimately you're going to need a Web server such as WebSphere Application Server or Apache's Tomcat. (If you're using WebSphere Studio V5 for development, you can skip this step until you're ready to put your application into production.)

Both of these servers are J2EE compliant, which means that the Web application you're building follows a particular directory structure so the server knows where to find everything. How you should get started depends on the environment you're using.

If you're using WebSphere Studio, the easiest way to start is to create a new Web Project. This provides you with a test environment against which you can run the servlets, and when you're ready to move to WebSphere Application Server, you can simply export the EAR file and deploy it as an Enterprise Application using the administration tools. (For more information on using WebSphere Studio and Application Server, see Resources.)

If you're using Tomcat, the situation is a bit simpler. All *applications* are located in the `webapps` directory. For example, after installation, Tomcat has a directory called `webapps/examples` in which the examples application lives. If you were to create a file called `test.html` and place it in this directory, you could access it by pointing your browser to:

```
http://localhost:8080/examples/test.html
```

Any servlets for this application are located in `webapps/examples/WEB-INF/classes` directory, which is mapped to the alias servlet. This means that to access the servlet `HelloWorldExample` (in the file `webapps/examples/WEB-INF/classes/HelloWorldExample.class`) you would point your browser to:

```
http://localhost:8080/examples/servlet/HelloWorldExample
```

When using Tomcat, the simplest way to create a new application is to copy one of the existing applications, rename the directory, and restart Tomcat.

Let's get started by creating a simple servlet and accessing it.

The basic servlet

If you're using WebSphere Studio, start by choosing **File => New => Project => Web Project** to create a new J2EE project, and name it `MultiUse`. (For simplicity's sake, place it in its own EAR file.) Choose **File => New => Other => Web => Servlet** to create a new servlet in the new project.

Create a new servlet named `XSLServlet`. (If you're not using WebSphere Studio, save it as a `.java` file in the `WEB-INF/classes` directory of your application.) Initially, it should look like this:

```
import java.io.IOException;
import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class XSLServlet extends HttpServlet {

    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

        resp.setContentType("text/html");
        java.io.PrintWriter out = resp.getWriter();
        out.print("<html>");
        out.print("<body>");
        out.print("<p>It works!</p>");
        out.print("</body>");
        out.print("</html>");

    }

    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

        doGet(req, resp);

    }

}
```

A servlet has many more methods than these (inherited from `HttpServlet`), but only `doGet()` and `doPost()` are important at the moment. The `doGet()` method is executed when a browser simply points at the servlet (making it easy to simulate by simply typing the URL), while the `doPost()` method is executed when a browser submits a form that uses `method="POST"`. In either case, the servlet should perform the same actions, so in the sample servlet, `doPost()` simply calls `doGet()`.

The `doGet()` method takes two arguments: the request and the response. The

`HttpServletRequest` object carries information on what type of browser the user has, what IP address he or she is coming from, and so on. It also carries any parameters that were submitted with the request.

The `HttpServletResponse` object actually sends the information to the user. When it does, that information needs to be identified as a certain type so the browser knows what to do with it. In this case, the application will serve a simple Web page, so set the `Content-Type` as `text/html`.

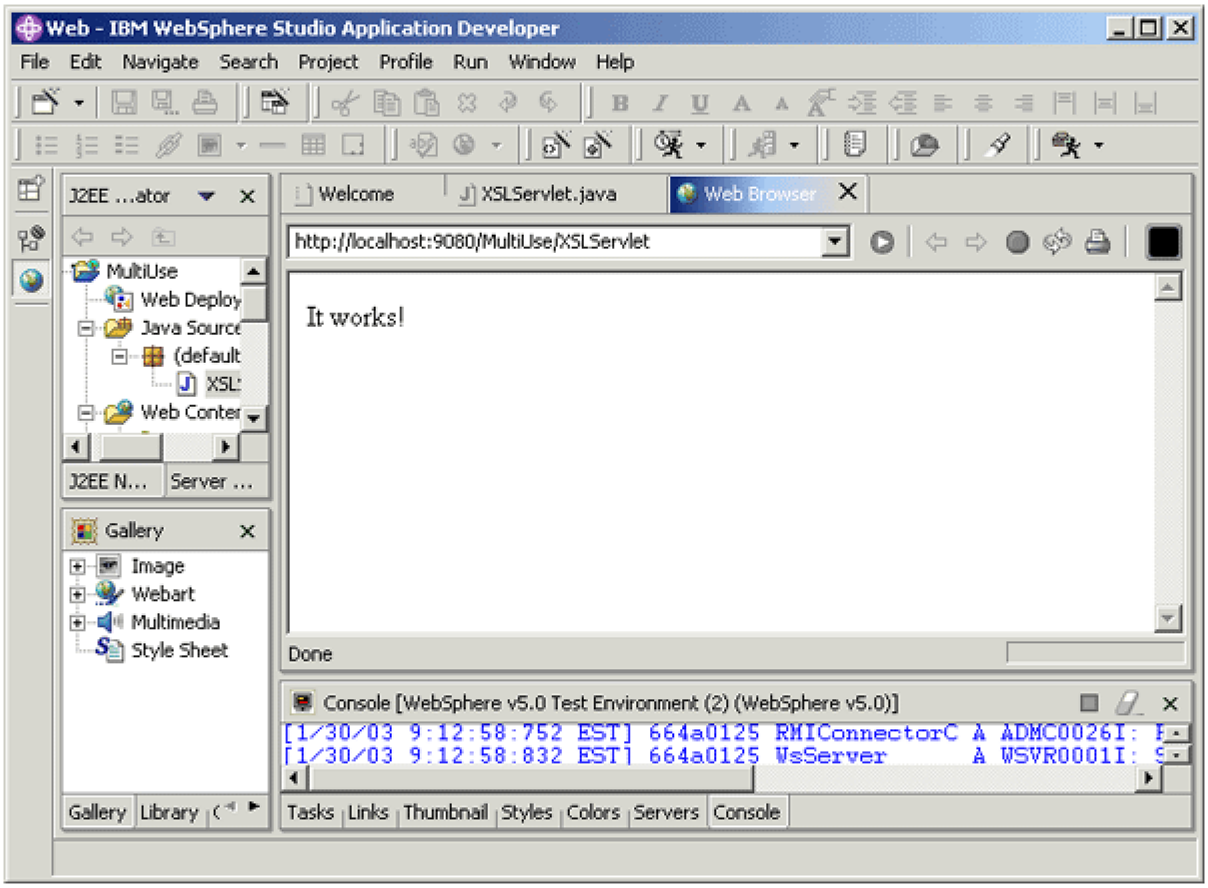
Finally, in order to actually output information, create a `PrintWriter` object by requesting it from the `HttpServletResponse` object, then use it to print.

Now let's see the results.

Compiling and accessing the servlet

If you're using WebSphere Studio, saving the `.java` file also compiles it and places the compiled class in the project's `Web Content/WEB-INF/classes` directory. To execute the servlet, simply right-click the servlet file and choose **Run on Server**. If asked, choose the **Test Environment**. If the server isn't running, start it and open a Web Browser window to display the results.

Figure 3. Web Browser window displaying servlet results


Web - IBM WebSphere Studio Application Developer
File Edit Navigate Search Project Profile Run Window Help
J2EE ...ator
MultiUse
Web Deploy
Java Source
(default
XSL
Web Conter
J2EE N... Server ...
Gallery
Image
Webart
Multimedia
Style Sheet
Gallery Library
Welcome
XSLServlet.java
Web Browser
http://localhost:9080/MultiUse/XSLServlet
Done
Console [WebSphere v5.0 Test Environment (2) (WebSphere v5.0)]
Tasks Links Thumbnail Styles Colors Servers Console

It works!

```
[1/30/03 9:12:58:752 EST] 664a0125 RMIConnectorC A ADMC0026I: F
[1/30/03 9:12:58:832 EST] 664a0125 WsServer      A WSVR0001I: S
```

If you're not using WebSphere Studio, compile the `XSLServlet.java` file, making sure that the `servlet.jar` file (typically found in Tomcat's `common\lib` directory) is on the classpath. Assuming an application named `MultiUse`, test the servlet by pointing your browser to:

```
http://localhost:8080/MultiUse/servlet/XSLServlet
```

Before moving on, let's make sure the phone simulator is working.

The mobile phone simulator

To run the mobile phone example, download the Openwave SDK from http://developer.openwave.com/download/product_62.html and install it. Part of this download is a simulator that enables a developer to see how a page will look and work on the small screen of a user's phone. It also enables the developer to catch XHTML that is not part of the XHTML Mobile specification.

To test the sample servlet on the phone, start the **Openwave SDK 6.2 HTTP** application and type the appropriate URL into the box next to **Go:**. For WebSphere Studio, use:

```
http://localhost:9080/MultiUse/XSLServlet
```

Because WebSphere Studio uses a plain HTTP server, it's available from outside the WebSphere Studio environment.

For Tomcat, use

```
http://localhost:8080/MultiUse/servlet/XSLServlet
```

as before.

Now that the servlet's in place, let's add the transformation.

Section 4. The basic transformation

The basic style sheet

Let's start with a simple transformation, using an arbitrary style sheet. The basic style sheet, intended for a desktop browser, displays a list of headlines and blurbs in most cases. It carries a parameter, `thisStory`, which can be used to indicate that only a single story should be displayed:

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

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
 <xsl:output method="html" indent="yes"/>

 <xsl:param name="thisStory" select="'all'"/>

 <xsl:template match="/">

   <html>
   <head><title>Vanguard Report headlines</title></head>
   <body>
     <center>
       <img
src="http://www.vanguardreport.com/phpnuke/themes/3D-Fantasy/images/logo.gif"
alt="logo" width="340" height="100" /></center>

     <h2 align="center">Today's Top Stories</h2>

     <table border="1">
       <xsl:if test="$thisStory='all'">
         <xsl:apply-templates select="/news/story"/>
       </xsl:if>
       <xsl:if test="$thisStory!='all'">
         <xsl:call-template name="singleStory"/>
```

```
        </xsl:if>
      </table>
    </body>
    </html>

  </xsl:template>
  <xsl:template match="story">
    <tr>
      <td>
        <xsl:element name="a">
          <xsl:attribute name="href">/MultiUse/XSLServlet?storyid=<xsl:value-of
            select="@storyid"/></xsl:attribute>
          <b><xsl:value-of select="headline" /></b>
        </xsl:element>
      </td>
      <td><xsl:value-of select="blurb" /></td>
    </tr>
  </xsl:template>

  <xsl:template name="singleStory">
    <h3><xsl:value-of select="/news/story[@storyid=$thisStory]/headline" /></h3>
    <p><xsl:value-of select="/news/story[@storyid=$thisStory]/blurb" /></p>
    <p>Get the full story
      <xsl:element name="a">
        <xsl:attribute name="href"><xsl:value-of
          select="/news/story[@storyid=$thisStory]/permalink"/></xsl:attribute>
          here.
      </xsl:element>
    </p>
  </xsl:template>


</xsl:stylesheet>
```

If you're using Tomcat, change the href attribute text from

```
/MultiUse/XSLServlet?storyid=
```

to

```
/MultiUse/servlet/XSLServlet?storyid=
```

so the generated link properly references the servlet.

Create the sources

Actually executing the transformation requires four basic steps:

1. Create the content source and the style source
2. Determine the destination for the result
3. Create the `Transformer` object using the style

4. Perform the transformation

First, create the sources:

```
...
import javax.xml.transform.stream.StreamSource;

public class XSLServlet extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {

    resp.setContentType("text/html");
    java.io.PrintWriter out = resp.getWriter();

   StreamSource contentSource =
                          new StreamSource("http://localhost:9080/MultiUse/news.xml");
                       StreamSource styleSource =
                          new
StreamSource("http://localhost:9080/MultiUse/browser.xsl");
    }
...
}
```

In the sample application, both the content and the style sheet are just files, but they can also be SAX streams, DOM documents, or other sources. Also, notice that both are specified here as fully-qualified URIs, so they can also be dynamically generated by another process.

Both of these URIs assume that you're using WebSphere Studio and that the files are in the `Web Content` directory. For Tomcat, put the files in the `MultiUse` directory and adjust the port number accordingly.

Determine the result

Like the content and style sheet sources, a transformation result can be a file, a DOM object, or a stream. In this case, the application sends the result of the transformation directly to the `PrintWriter` that outputs content to the browser.

```
...
import javax.xml.transform.stream.StreamResult;

public class XSLServlet extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {

    resp.setContentType("text/html");
    java.io.PrintWriter out = resp.getWriter();

    StreamSource contentSource =
          new StreamSource("http://localhost:9080/MultiUse/news.xml");
    StreamSource styleSource =
          new StreamSource("http://localhost:9080/MultiUse/browser.xsl");
```

```
                          StreamResult result = new StreamResult(out);

    }
...
}
```

Now that the sources and result have been determined, it's time to create the actual `Transformer` object.

Create the Transformer

The `Transformer` object actually performs the XSL transformation, and in order to do that, it takes its instructions from the style sheet source:

```
...
import javax.xml.transform.TransformerFactory;
                    import javax.xml.transform.Transformer;
                    import javax.xml.transform.TransformerConfigurationException;

public class XSLServlet extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {

    resp.setContentType("text/html");
    java.io.PrintWriter out = resp.getWriter();

      try{
        StreamSource contentSource =
          new StreamSource("http://localhost:9080/MultiUse/news.xml");
        StreamSource styleSource =
          new StreamSource("http://localhost:9080/MultiUse/browser.xsl");

        StreamResult result = new StreamResult(out);

        TransformerFactory transformerFactory =
                             TransformerFactory.newInstance();
                          Transformer transformer =
                             transformerFactory.newTransformer(styleSource);
                        } catch (TransformerConfigurationException e) {
                          out.print("TransformerConfigurationException:
"+e.getMessage());
                        }
  }
...
}
```

The `TransformerFactory` creates the `Transformer` using the style sheet source as input. To perform an identity transform, in which the XML data remains unchanged, simply create the `Transformer` with no style sheet input. This technique acts as a handy serialization tool for situations where you might want to, say, output a DOM document.

Now the application is ready to perform the actual transformation.

Perform the transformation

Actually performing the transformation is straightforward. Simply call the `transform()` method, providing a source to transform and a result to receive the transformed document. (Remember, the style sheet was used to create the `Transformer` object itself.)

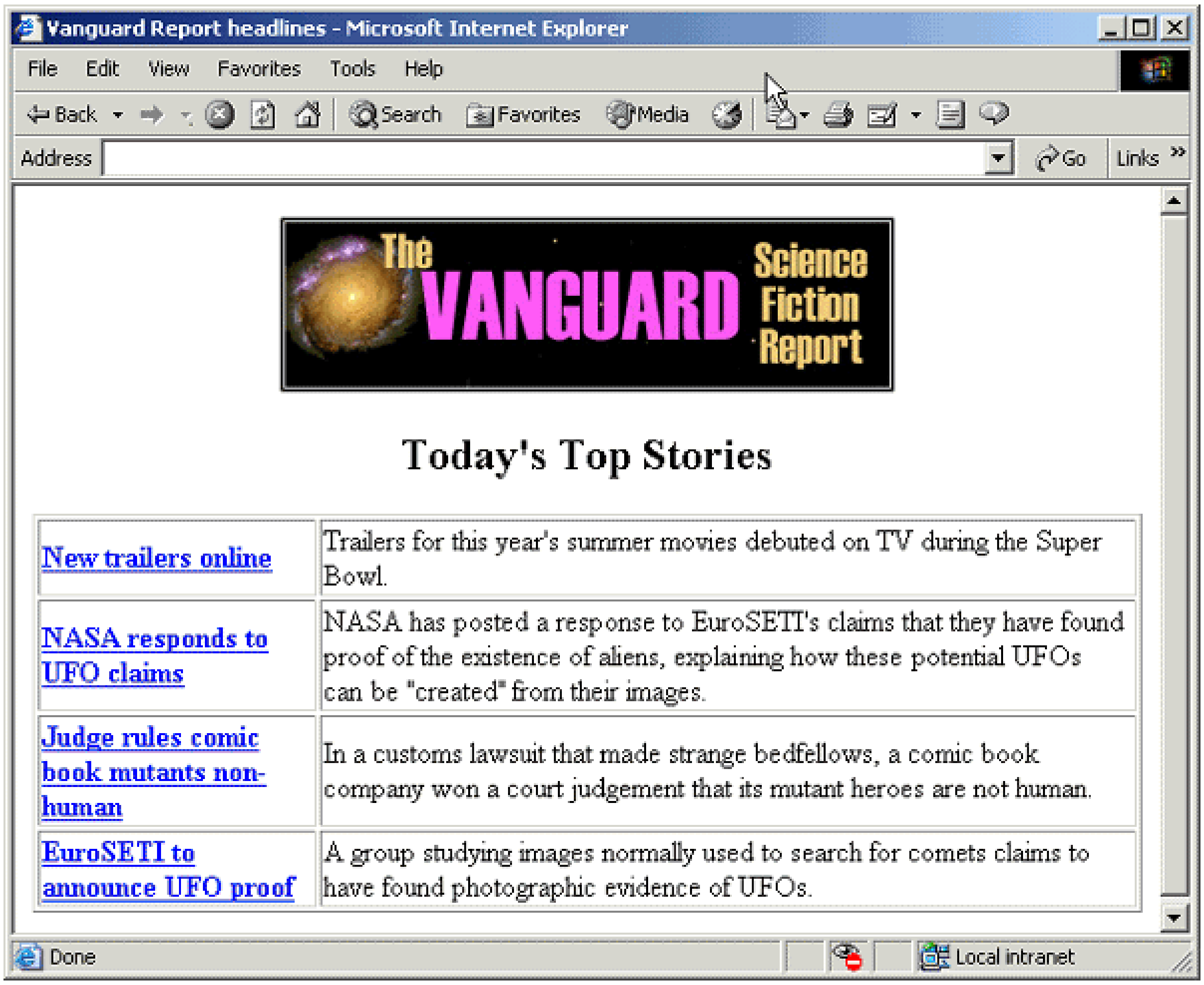
```
import java.io.IOException;
import javax.servlet.ServletException;

...
import javax.xml.transform.TransformerException;

public class XSLServlet extends HttpServlet {

...
           Transformer transformer =
                        transformerFactory.newTransformer(styleSource);

           transformer.transform(contentSource, result);

        } catch (TransformerConfigurationException e) {
           out.print("TransformerConfigurationException: "+e.getMessage());
        } catch (TransformerException e) {
                                  out.print("TransformerException:"+e.getMessage());
        }

    }
...
}
```

The transformer takes the `contentSource`, transforms it, and sends the new structure to the `result` object.

See the results

To see the results, make sure the news.xml and browser.xsl files are in their proper locations and execute the servlet as discussed in Compiling and accessing the servlet.

Figure 4. Basic output page for traditional desktop browsers


Vanguard Report headlines - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address Go Links
The VANGUARD Science Fiction Report
Done
Local intranet

Today's Top Stories

New trailers online	Trailers for this year's summer movies debuted on TV during the Super Bowl.
NASA responds to UFO claims	NASA has posted a response to EuroSETI's claims that they have found proof of the existence of aliens, explaining how these potential UFOs can be "created" from their images.
Judge rules comic book mutants non-human	In a customs lawsuit that made strange bedfellows, a comic book company won a court judgement that its mutant heroes are not human.
EuroSETI to announce UFO proof	A group studying images normally used to search for comets claims to have found photographic evidence of UFOs.

Retrieving a request parameter

So far the servlet processes the style sheet as-is, outputting all of the potential stories, but the style sheet itself is designed to allow for the addition of a parameter that determines which story to display. This parameter is passed as part of the URL when the user clicks on one of the links created on the page.

The first step is to retrieve this information from the `storyid` parameter submitted with the original request:

```
...
           Transformer transformer =
                        transformerFactory.newTransformer(styleSource);

           String thisStory = req.getParameter("storyid");

           transformer.transform(contentSource, result);

        } catch (TransformerConfigurationException e) {
           out.print("TransformerConfigurationException: "+e.getMessage());
```

...

The next step is to set the parameter on the style sheet.

Setting a style sheet parameter

To set the parameter for the style sheet itself, use the `Transformer` object's `setParameter()` method:

```
...
           String thisStory = req.getParameter("storyid");
           if (thisStory == null) {
                                      thisStory = "all";
                                }
                                transformer.setParameter("thisStory", thisStory);

           transformer.transform(contentSource, result);

        } catch (TransformerConfigurationException e) {
           out.print("TransformerConfigurationException: "+e.getMessage());
...
```

The style sheet itself is built so that if no parameter is submitted, `thisStory` defaults to `all`; thus, if no parameter has been submitted with the request, set `thisStory` to `all` to match that expectation.

Section 5. Selecting a style sheet

Pointing to the style sheet

So far, the application has been using an arbitrary style sheet, but in the final application, the XML file will have several possible style sheets noted within it. The servlet will then check each one against the request to decide which to use.

Adding the style sheet directive to the XML file is a straightforward matter of adding the xml-stylesheet processing instruction:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet
     href="http://localhost:9080/MultiUse/browser.xsl"
                         type="text/xsl" ?>

<news>
   <story storyid="1">
       <headline>New trailers online</headline>
```

```
        <blurb>Trailers for this year's summer movies debuted
              on TV during the Super Bowl.</blurb>
...
```

With the directive in place, the servlet can extract the information at run-time.

Getting the style sheet

When retrieving the style sheet information from the XML file, the only thing that changes is the designation of the style sheet `Source` object:

```
...
import javax.xml.transform.Source;

public class XSLServlet extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {

    resp.setContentType("text/html");
    java.io.PrintWriter out = resp.getWriter();

    try{
      StreamSource contentSource =
          new StreamSource("http://localhost:9080/MultiUse/news.xml");

      StreamResult result = new StreamResult(out);

      TransformerFactory transformerFactory =
                     TransformerFactory.newInstance();

      Source styleSource =
                                       transformerFactory.getAssociatedStylesheet(
                                       contentSource, null, null ,null);

      Transformer transformer =
                        transformerFactory.newTransformer(styleSource);

      String thisStory = req.getParameter("storyid");
      if (thisStory == null) {
                  thisStory = "all";
      }
      transformer.setParameter("thisStory", thisStory);

      transformer.transform(contentSource, result);
...
}
```

Note that the `TransformerFactory` retrieves the style sheet so that it's available to create the `Transformer`.

In this case, the servlet is retrieving a single style sheet -- you can test it by executing the servlet -- but in the application in progress here, the XML file will have several choices available. The three null values in `getAssociatedStyleSheet()` are involved in making that choice, as seen in Choosing based on the media.

The mobile style sheet

The second style sheet that's added to the mix is the *mobile* style sheet, for smaller devices such as cell phones and PDAs:

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

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
 <xsl:output method="html" indent="yes"/>

 <xsl:param name="thisStory" select="'all'"/>

 <xsl:template match="/">

   <html>
   <head><title>Vanguard Report Top Stories</title></head>
   <body>
     <ul>
       <xsl:if test="$thisStory='all'">
          <xsl:apply-templates select="/news/story"/>
       </xsl:if>
       <xsl:if test="$thisStory!='all'">
          <xsl:call-template name="singleStory"/>
       </xsl:if>
     </ul>
   </body>
   </html>

 </xsl:template>
 <xsl:template match="story">
   <li>
     <xsl:element name="a">
       <xsl:attribute
          name="href">/MultiUse/XSLServlet?storyid=<xsl:value-of
          select="@storyid"/></xsl:attribute>
       <xsl:value-of select="headline" />
     </xsl:element>
   </li>
 </xsl:template>

 <xsl:template name="singleStory">
   <p><b><xsl:value-of
       select="/news/story[@storyid=$thisStory]/headline" /></b></p>
   <p>
     <xsl:element name="a">
       <xsl:attribute name="href"><xsl:value-of
          select="/news/story[@storyid=$thisStory]/permalink"/></xsl:attribute>
       Full Story
     </xsl:element>
   </p>
 </xsl:template>

</xsl:stylesheet>
```

You can reference this style sheet in the XML file just as you referenced the first one.

Adding an alternate style sheet

XSL Transformations are specifically geared towards taking the same content and repurposing it for different media. Specifically, a style sheet for the Web is going to look very different from a style sheet that prepares content for output as a paper brochure. For this purpose, the style sheet directive can carry a `media` attribute to distinguish between style sheets for different purposes.

Because there is no set of required values, however, you can arbitrarily set the media attribute to distinguish between different presentations as you see fit. For example, you could have different style sheets for older versions of Netscape versus current versions, or for Netscape versus Internet Explorer.

In the case of this tutorial, you'll specify a `general` style sheet and a `mobile` style sheet:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="http://localhost:9080/MultiUse/mobile.xsl"
    type="text/xsl"
    media="mobile" alternate="yes"?>
<?xml-stylesheet href="http://localhost:9080/MultiUse/browser.xsl"
    type="text/xsl"
    media="general"?>

<news>
   <story storyid="1">
       <headline>New trailers online</headline>
       <blurb>Trailers for this year's summer movies debuted on TV
       during the Super Bowl.</blurb>
...
```

The servlet can then use this information to choose the appropriate style sheet.

Choosing based on the media

Using the `media` parameter, choose the `general` style sheet.

```
...
    TransformerFactory transformerFactory =
                 TransformerFactory.newInstance();

    Source styleSource =
                 transformerFactory.getAssociatedStylesheet(contentSource,
                 "general", null ,null);

    Transformer transformer =
                 transformerFactory.newTransformer(styleSource);
...
```

Here the `TransformerFactory` is looking for the style sheet with a `media` value of `general`. The other two values enable you to choose a style sheet based on the optional `title` and `charset` attributes.

Here the choice is arbitrary; next you'll base it on the user-agent.

Choosing based on the user-agent

When you surf the Web, your browser gives the Web server a number of different information items about you, including the IP address you're surfing from and the type of browser you're using. This information is sent to the server in the form of headers that are part of the request.

The type of browser, or user agent, can be seen in the USER-AGENT header. For example, the machine on which this tutorial is being written has three different browsers installed:

Netscape Navigator 4.790

```
Mozilla/4.79 [en] (Windows NT 5.0; U)
```

Microsoft Internet Explorer 6.0

```
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
```

Openwave SDK v6.2 HTTP

```
OPWV-SDK/62 UP.Browser/6.2.0.1.185 (GUI) MMP/2.0
```

Notice that each is unique, so it's possible to use this information to decide which style sheet `media` value to use:

```
...
public class XSLServlet extends HttpServlet {

    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

        resp.setContentType("text/html");
        java.io.PrintWriter out = resp.getWriter();

        String media = "";
                            if (req.getHeader("USER-AGENT").indexOf("UP.") > 0) {
                                media = "mobile";
                            } else {
                                media = "general";
                            }

        try{
           StreamSource contentSource =
              new StreamSource("http://localhost:9080/MultiUse/news.xml");

           StreamResult result = new StreamResult(out);

           TransformerFactory transformerFactory =
                     TransformerFactory.newInstance();

           Source styleSource =
                     transformerFactory.getAssociatedStylesheet(contentSource,
media
, null ,null);

           Transformer transformer =
                     transformerFactory.newTransformer(styleSource);
...
```

Now, if you run the servlet in the browser and in the phone simulator, you should see different results. (Note that the phone simulator has to be manually refreshed. Click the **M** button, then the **8** key to reload the page.)

Of course, it would get extremely tedious to hard-code every possible combination of browser agents into the application itself. Instead, you can create a list of criteria in a properties file, which the servlet can easily read and understand.

Creating properties

A properties file is simply a text file with name-value pairs, one pair per line, as in:

```
Mozilla=general
MSIE=general
UP.=mobile
Nokia=mobile
```

Note that a properties file can use an equals sign (=), a colon (`:`), or a plain white space as a delimeter between a key and its value.

Create a properties file and save it anywhere on your system, as long as it's accessible by the servlet.

Using properties

The first step in using a properties file is to create a `Properties` object and load the new file:

```
...
import java.util.Properties;
                    import java.io.InputStream;
                    import java.io.FileNotFoundException;

public class XSLServlet extends HttpServlet {

 public void doGet(HttpServletRequest req, HttpServletResponse resp)
  throws ServletException, IOException {

  resp.setContentType("text/html");
  java.io.PrintWriter out = resp.getWriter();
  String media = getMedia(req.getHeader("USER-AGENT"));

  try{
    StreamSource contentSource =
       new StreamSource("http://localhost:9080/MultiUse/news.xml");
...
 public static String getMedia(String agent){

                      String media = "general";

                      Properties browserProps = new Properties();
                      try {
                       browserProps.load(new
java.io.FileInputStream("h:\\browser.properties"));
                      } catch (FileNotFoundException e) {
                       e.printStackTrace();
                      } catch (IOException e) {
                       e.printStackTrace();
                      }

                      return media;
                     }
}
```

In this case, the file `browser.properties` was located at the root of the h: drive. The double backslash (`\\`) is included so that Java understands you're looking for a backslash and not trying to escape the b in `browser`.

Searching properties

All that's left is to have the `getMedia()` method search each of the available properties to see if any of them match the user-agent.

```
...
import java.util.Enumeration;

public class XSLServlet extends HttpServlet {
...
 public static String getMedia(String agent){

   String media = "general";

   Properties browserProps = new Properties();
   try {
       browserProps.load(new java.io.FileInputStream("h:\\browser.properties"));
       Enumeration allBrowsers = browserProps.propertyNames();
                           while (allBrowsers.hasMoreElements()){
                             String thisBrowser = allBrowsers.nextElement().toString();
                             if (agent.indexOf(thisBrowser) > 0) {
                                 media = browserProps.getProperty(thisBrowser);
                                 break;
                             }
                           }
   } catch (FileNotFoundException e) {
            e.printStackTrace();
   } catch (IOException e) {
            e.printStackTrace();
   }

   return media;
 }

}
```

The `propertyNames()` method returns an enumeration of all the keys (such as, in this case, `Mozilla` or `Up.`). The servlet can then run through each of those names, checking to see whether it's part of the user-agent. If it does find one, it sets the media value and breaks out of the loop.

In this way, you can add new style sheets and new presentation platforms without ever having to change the servlet itself. In fact, by altering the servlet to take the XML file as a parameter, you can create a servlet that is completely generic.

Section 6. The servlet as a Web service

The view from 10,000 feet

The purpose of this tutorial is to demonstrate the use of XSLT to make content available in a number of different contexts, so let's take a moment to look at a situation that is different from a typical browser setup.

In the emerging world of Web services, a system sends a specially formed XML message, known as a SOAP message, to a server, which interprets it, generates another SOAP message in response, and sends it back to the original requester. The ultimate goal is typically a single piece of data or a group of data items.

Creating an actual Web service can be difficult, but fortunately WebSphere Studio automates the entire process, creating a **proxy** that acts as a wrapper for a class such as the servlet. The proxy receives the SOAP message and decodes it, determining what method is being requested. It then executes the method and encodes the result in a SOAP response.

To see how this would work, you can wrap the servlet in a Web service and then test it.

Moving the transformation

Before you can actually wrap the servlet, however, you need to make a change to it. A Web service returns a piece of information in the form of a `String` or other object, so the stream of information returned by the `HttpServletResponse` object isn't ideally suited for it. Fortunately, all that's being returned is the `String` value of the transformation, so by breaking it out into a separate method, you can use it for both purposes:

```
...
```

```
import java.io.StringWriter;

public class XSLServlet extends HttpServlet {

 public static String getTransform(String media, String thisStory){
                       try{
                          StreamSource contentSource =
                             new StreamSource("http://localhost:9080/MultiUse/news.xml");

                          StringWriter resultStringWriter = new StringWriter();
                          StreamResult result = new StreamResult(resultStringWriter);

                          TransformerFactory transformerFactory =
                                          TransformerFactory.newInstance();

                          Source styleSource =
                         transformerFactory.getAssociatedStylesheet(
                         contentSource, media, null ,null);

                          Transformer transformer =
                                       transformerFactory.newTransformer(styleSource);

                          if (thisStory == null) {
                                      thisStory = "all";
                          }
                          transformer.setParameter("thisStory", thisStory);

                          transformer.transform(contentSource, result);

                          return resultStringWriter.toString();

                       } catch (TransformerConfigurationException e) {
                          return "TransformerConfigurationException: "+e.getMessage();
                       } catch (TransformerException e) {
                          return "TransformerException:"+e.getMessage();
                       }

                        }

 public void doGet(HttpServletRequest req, HttpServletResponse resp)
   throws ServletException, IOException {

   resp.setContentType("text/html");
   java.io.PrintWriter out = resp.getWriter();

   String media = getMedia(req.getHeader("USER-AGENT"));
                       String thisStory = req.getParameter("storyid");

                       out.print(getTransform(media, thisStory));
 }
...
```

Aside from the reorganization, the only thing that's really changed here is that instead of sending the result to the `PrintWriter`, the method sends the result to a `StringWriter`. The `StringWriter` is a handy class that enables you to save as a `String` information that normally would have been immediately written to a stream.

When the transformation is complete, the `String` is returned. In the case of a typical request, `doGet()` calls `getTransform()` and returns the `String` to the browser. However, there's nothing to say that `getTransform()` can't be called directly. That's what will happen with the Web service.

The Web services style sheet

Before you go any farther, you need to create the new style sheet for the Web service:

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

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
  <xsl:output method="text" indent="yes"/>

  <xsl:param name="thisStory" select="'all'"/>

  <xsl:template match="/">
      <xsl:if test="$thisStory='all'">
         <xsl:apply-templates select="/news/story"/>
      </xsl:if>
      <xsl:if test="$thisStory!='all'">
         <xsl:call-template name="singleStory"/>
      </xsl:if>
  </xsl:template>
  <xsl:template match="story">
      <xsl:value-of select="headline" /><xsl:text>
</xsl:text>
  </xsl:template>

  <xsl:template name="singleStory">
      <xsl:value-of select="/news/story[@storyid=$thisStory]/blurb" />
  </xsl:template>

</xsl:stylesheet>
```

The style sheet is simple, returning a sequence of headlines or a single blurb, depending on whether a particular story has been requested.

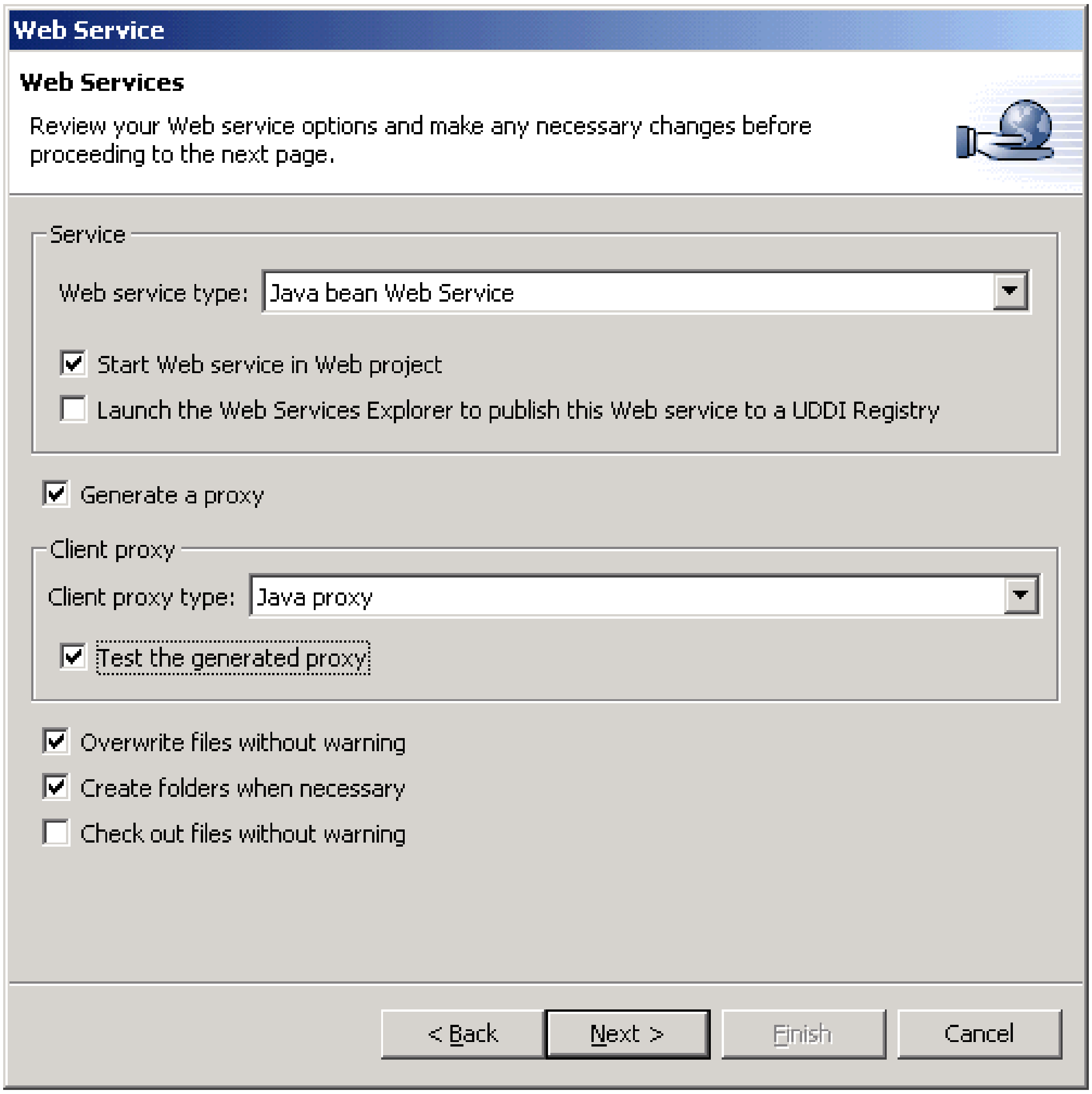
The new style sheet is also added to the XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="http://localhost:9080/MultiUse/webservice.xsl"
type="text/xsl"
media="webservice" alternate="yes"?>
<?xml-stylesheet href="http://localhost:9080/MultiUse/mobile.xsl"
type="text/xsl"
media="mobile" alternate="yes"?>
<?xml-stylesheet href="http://localhost:9080/MultiUse/browser.xsl"
type="text/xsl"
media="general" ?>

<news>
...
```

Create the proxy service

To create the proxy, choose **File => New => Other => WebServices => Web Service => Next**. For the **Web service type**, choose **Java bean Web Service**. Make sure **Generate a proxy** is selected, and select **Test the generated proxy**.

Figure 5. Service generation dialog



Web Service

Web Services

Review your Web service options and make any necessary changes before proceeding to the next page.

Service

Web service type: Java bean Web Service

[✓] Start Web service in Web project

[ ] Launch the Web Services Explorer to publish this Web service to a UDDI Registry

[✓] Generate a proxy

Client proxy

Client proxy type: Java proxy

[✓] Test the generated proxy

[✓] Overwrite files without warning

[✓] Create folders when necessary

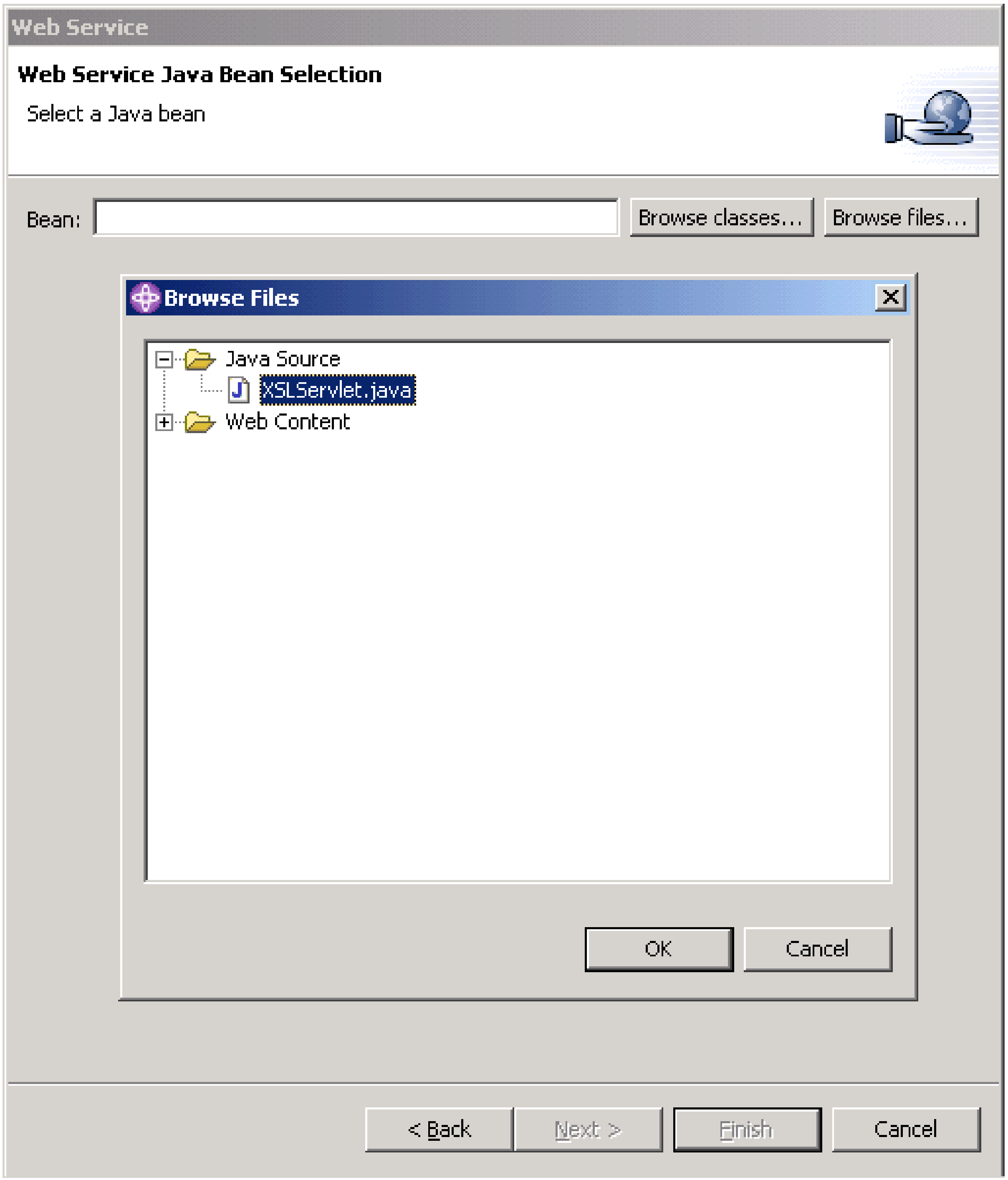
[ ] Check out files without warning

Click **Next**, and then click **Next** again to use the default server environment.

Create the proxy service, continued

Click **Browse files...** and navigate to the servlet.

Figure 6. Choosing the servlet



Click **OK** to select the class, and then click **Finish** to create the Web service.

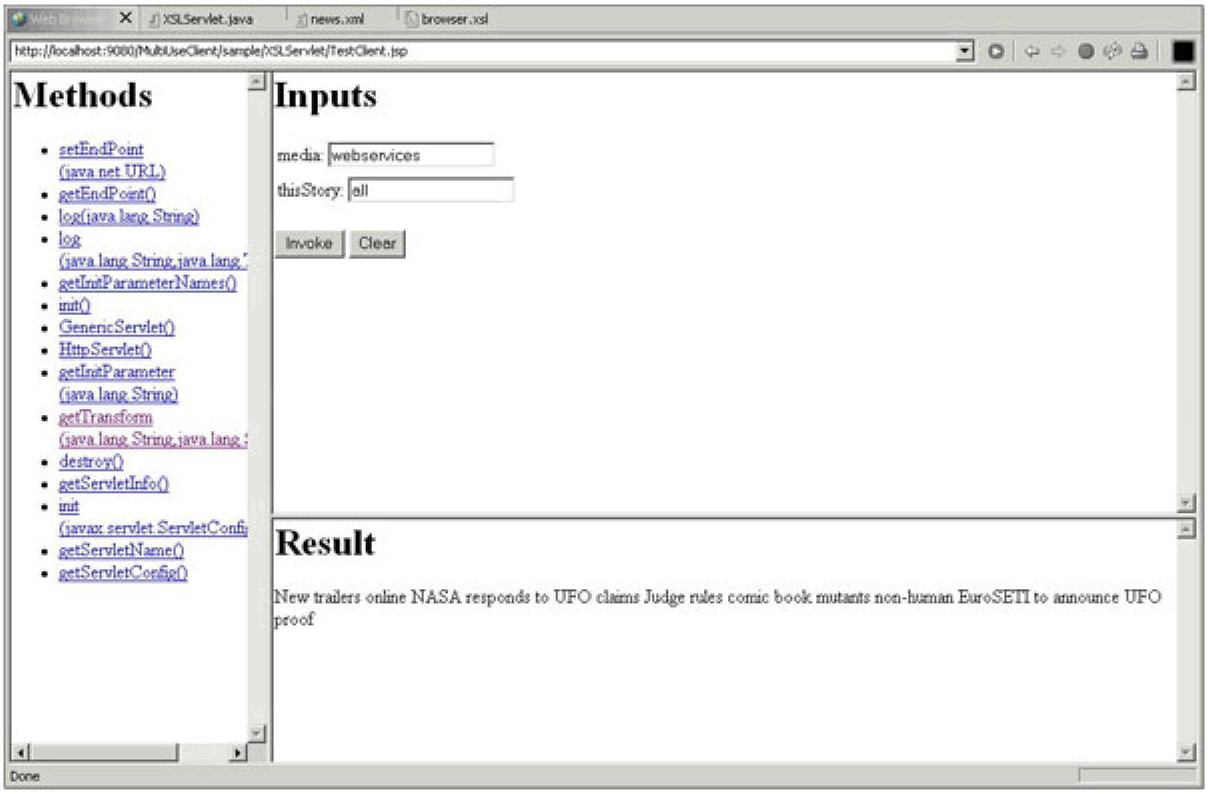
WebSphere Studio takes a minute or two to generate the Web service and a sample client with which you can test it.

Test the service

WebSphere Studio automatically brings up a series of Web pages that enable you to test the new Web service. Down the left side you will see all of the methods available. Because we didn't narrow it down when creating the service, all of the methods for a servlet are available to the client. Scroll this frame until you can click the `getTransform()` link.

Click this link to bring up a form in the top right frame. This form gives you an opportunity to enter values for the two parameters of `getTransform()`, which are, as you may recall, the `media` and the storyid, `thisStory`. Enter `webservices` as the media and enter `all` for `thisStory`. Click `Invoke` to see the results in the bottom frame.

Figure 7. Page results for Web services


Web Browser
XSLServlet.java
news.xml
browser.xsl
http://localhost:9080/MultiUseClient/sample/XSLServlet/TestClient.jsp
Methods
setEndPoint (java.net.URL)
getEndPoint()
log(java.lang.String)
log (java.lang.String,java.lang.
getInitParameterNames()
init()
GenericServlet()
HttpServlet()
getInitParameter (java.lang.String)
getTransform (java.lang.String,java.lang.
destroy()
getServletInfo()
init (javax.servlet.ServletConfi
getServletName()
getServletConfig()
Inputs
Invoke
Clear
Result
Done

media: webservices

thisStory: all

New trailers online NASA responds to UFO claims Judge rules comic book mutants non-human EuroSETI to announce UFO proof

Section 7. Summary

Tutorial summary

XSL Transformations were created to enable the same content to be rendered in multiple ways for different purposes. This tutorial has demonstrated the ways that you can use a Java servlet to distinguish between different requesters and return the appropriate content based on the appropriate style sheet. In this tutorial, you have learned how to:

- Execute a transformation
- Indicate a style sheet on an XML document
- Choose between different style sheets
- Use properties to choose from many different possibilities
- Use XSLT in conjunction with a Web service

Resources

Learn

- Get a better understanding of the XSL Transformations process with:
    - Introduction to XML (*developerWorks*, August 2002)
    - XML Programming in Java (*developerWorks*, September 1999)
    - Manipulating data with XSL (*developerWorks*, October 2001)
- Learn about servlets with the Building Java HTTP servlets tutorial (*developerWorks*, September 2000).
- Take the Introduction to WebSphere Studio tutorial to familiarize yourself with development using the WebSphere Studio platform in general (*developerWorks*, October 2000).
- Get a feel for Developing database applications with WebSphere and DB2.
- For a look at deploying applications to WebSphere Application Server, take the Building Web Services with WebSphere Studio Part 2: Deploy and publish tutorial.
- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- Download Sun's Java 2 Standard Edition version 1.4 at http://java.sun.com/j2se/1.4/.
- Download Apache Tomcat at http://tomcat.apache.org/tomcat-4.1-doc/.
- Build your next development project with IBM trial software, available for download directly from developerWorks.

Discuss

- Participate in the discussion forum for this content.

About the author

Nicholas Chase
Nicholas Chase, a Studio B author, has been involved in Web site development for companies such as Lucent Technologies, Sun Microsystems, Oracle, and the Tampa Bay Buccaneers. Nick has been a high school physics teacher, a low-level radioactive waste facility manager, an online science fiction magazine editor, a multimedia engineer, and an Oracle instructor. More recently, he was the Chief

Technology Officer of Site Dynamics Interactive Communications in Clearwater, Florida, USA, and is the author of four books on Web development, including *XML Primer Plus* (Sams). He loves to hear from readers and can be reached at nicholas@nicholaschase.com.