Servlets

How to use FOP in a Servlet

1. Overview

This page discusses topic all around using FOP in a servlet environment.

2. Example Servlets in the FOP distribution

In the directory {fop-dir}/examples/servlet, you'll find a working example of a FOP-enabled servlet.

You can build the servlet easily by using the supplied Ant script. After building the servlet, drop fop.war into the webapps directory of Tomcat. Then, you can use URLs like the following to generate PDF files:

- http://localhost:8080/fop/fop?fo=/home/path/to/fofile.fo
- http://localhost:8080/fop/fop?xml=/home/path/to/xmlfile.xml&xsl=/home/path/to/xslfile.xsl

The source code for the servlet can be found under {fop-dir}/examples/servlet/src/FopServlet.java.

3. Create your own Servlet

Note:

This section assumes you are familiar with embedding FOP.

3.1. A minimal Servlet

Here is a minimal code snippet to demonstrate the basics:

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response) throws ServletException {
    try {
        response.setContentType("application/pdf");
        Driver driver = new Driver(new InputSource("foo.fo"),
                                   response.getOutputStream());
        driver.setRenderer(Driver.RENDER_PDF);
```

```
            driver.run();
        } catch (Exception ex) {
            throw new ServletException(ex);
        }
}
```

Note:

There are numerous problems with the code snippet above. Its purpose is only to demonstrate the basic concepts. See below for details.

3.2. Adding XSL tranformation (XSLT)

A common requirement is the to transform an XML source to XSLFO using an XSL transformation. It is recommended to use JAXP for this task. The following snippet shows the basic code:

```
protected Logger log;
protected TransformerFactory transformerFactory;

public void init() throws ServletException {
    this.log = new ConsoleLogger(ConsoleLogger.LEVEL_WARN);
    this.transformerFactory = TransformerFactory.newInstance();
}

[..]

    //Setup FOP
    Driver driver = new Driver();
    driver.setLogger(this.log);
    driver.setRenderer(Driver.RENDER_PDF);

    //Setup a buffer to obtain the content length
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    driver.setOutputStream(out);

    //Setup Transformer
    Source xsltSrc = new StreamSource(new File("foo-xml2fo.xsl"));
    Transformer transformer = this.transformerFactory.newTransformer(xsltSrc);

    //Make sure the XSL transformation's result is piped through to FOP
    Result res = new SAXResult(driver.getContentHandler());

    //Setup input
    Source src = new StreamSource(new File("foo.xml"));

    //Start the transformation and rendering process
    transformer.transform(src, res);

    //Prepare response
    response.setContentType("application/pdf");
```

```
    response.setContentLength(out.size());

    //Send content to Browser
    response.getOutputStream().write(out.toByteArray());
    response.getOutputStream().flush();
```

Note:

Buffering the generated PDF in a ByteArrayOutputStream is done to avoid potential problems with the Acrobat Reader Plug-in in IEx.

The `Source` instance used above is simply an example. If you have to read the XML from a string, supply a `new StreamSource(new StringReader(xmlstring))`. Constructing and reparsing an XML string is generally less desirable than using a SAXSource if you generate your XML. You can alternatively supply a DOMSource as well. You may also use dynamically generated XSL if you like.

Because you have an explicit `Transformer` object, you can also use it to explicitly set parameters for the transformation run.

3.3. Custom configuration

If you need to supply a special configuration do this in the `init()` method so it will only be done once and to avoid multithreading problems.

```
public void init() throws ServletException {
    [..]
    new Options(new File("userconfig.xml"));
    //or
    Configuration.put("baseDir", "/my/base/dir");
}
```

3.4. Improving performance

There are several options to consider:

- Instead of java.io.ByteArrayOutputStream consider using the ByteArrayOutputStream implementation from the Jakarta Commons IO project which allocates less memory.
- In certain cases it can help to write the generated PDF to a temporary file so you can quickly reuse the file. This is especially useful, if Internet Explorer calls the servlet multiple times with the same request or if you often generate equal PDFs.

Of course, the performance hints from the Embedding page apply here, too.

4. Notes on Microsoft Internet Explorer

Some versions of Internet Explorer will not automatically show the PDF or call the servlet

multiple times. These are well-known limitations of Internet Explorer and are not a problem of the servlet. However, Internet Explorer can still be used to download the PDF so that it can be viewed later. Here are some suggestions in this context:

- Use an URL ending in `.pdf`, like `http://myserver/servlet/stuff.pdf`. Yes, the servlet can be configured to handle this. If the URL has to contain parameters, try to have **both** the base URL as well as the last parameter end in `.pdf`, if necessary append a dummy parameter, like `http://myserver/servlet/stuff.pdf?par1=a&par2=b&d=.pdf`. The effect may depend on IEx version.
- Give IEx the opportunity to cache. In particular, ensure the server does not set any headers causing IEx not to cache the content. This may be a real problem if the document is sent over HTTPS, because most IEx installations will by default *not* cache any content retrieved over HTTPS. Setting the `Expires` header entry may help in this case:
```
response.setDateHeader("Expires", System.currentTimeMillis()
+ cacheExpiringDuration * 1000);
```
Consult your server manual and the relevant RFCs for further details on HTTP headers and caching.
- Cache in the server. It may help to include a parameter in the URL which has a timestamp as the value min order to decide whether a request is repeated. IEx is reported to retrieve a document up to three times, but never more often.

5. Servlet Engines

When using a servlet engine, there are potential CLASSPATH issues, and potential conflicts with existing XML/XSLT libraries. Servlet containers also often use their own classloaders for loading webapps, which can cause bugs and security problems.

5.1. Tomcat

Check Tomcat's documentation for detailed instructions about installing FOP and Cocoon. There are known bugs that must be addressed, particularly for Tomcat 4.0.3.

5.2. WebSphere 3.5

Put a copy of a working parser in some directory where WebSphere can access it. For example, if /usr/webapps/yourapp/servlets is the CLASSPATH for your servlets, copy the Xerces jar into it (any other directory would also be fine). Do not add the jar to the servlet CLASSPATH, but add it to the CLASSPATH of the application server which contains your web application. In the WebSphere administration console, click on the "environment" button in the "general" tab. In the "variable name" box, enter "CLASSPATH". In the "value"

box, enter the correct path to the parser jar file (/usr/webapps/yourapp/servlets/Xerces.jar in our example here). Press "OK", then apply the change and restart the application server.

6. Handling complex use cases

Sometimes the requirements for a servlet get quite sophisticated: SQL data sources, multiple XSL transformations, merging of several datasources etc. In such a case consider using Apache Cocoon instead of a custom servlet to accomplish your goal.