Back to Article

Using FOP With Java - Part 3
Graphics with FOP

Introduction

Last month, we saw how to implement a Print solution to print the FOP generated documents. We have also seen how a windows default printer can be used and how the output can directly be piped into a UNIX print queue. So far, we have discussed how to produce text based documents. However, in real life documents often need to embed images and graphics. In this article, we will examine the graphics capability of the FOP solution.

FOP has a wide range of graphics support. In short, it supports the following image formats:

- BMP (Windows Bitmap).
- EPS (Encapsulated Postscript).
- GIF (Graphics Interchange Format).
- JPEG (Joint Photographic Expert Group).
- PNG (Portable Network Graphics).
- SVG (Scalar Vector Graphics).
- TIFF (Tag Image File Format).

All the above mentioned image formats are reasonably well- supported within FOP. However, different image formats suffer from limitations depending upon the generated output document formats. These limitations are in the areas of resolution of images, the scaling of images, color space and color profiles. These limitations are the only demerit, if you're really dealing with advanced graphics. For day to day use, the most commonly used formats are color spaces and color profiles. These are very well supported by FOP. In this article, I will not try to focus on the limitations but focus on how we can use the most commonly used formats of graphics with FOP Generated documents.

Using Graphics from External Resource

XSL-FO provides a mechanism to load graphics from external resources. These external resources are typically an image file residing in the local or remote address. The source image is specified by the src attribute whose value is a URI. It handles HTTP, FYP data and file system resource locators in URI format. An unqualified URI is treated as a path to a file in the local file system. If the path is relative, it is calculated from the location of the source XSL-FO document. For example, you can refer to an image resource by c:\images\test.gif (absolute path) or you can access the same resource by specifying "test.gif" so long as the \images directory is present in the source XSL-FO document path. You can also access the same image over HTTP as http://server/images/test.gif . The important thing is that the resource should be available at the time of FOP transformation. This is crucial as after generating the document, you can decide to delete or move the image resource.

In general, we will load an external graphics in XSL-FO by using the <fo:external-graphic> element. The <fo:external-graphic> flow object is used for a graphic where the graphics data resides outside the fo:element tree. This element supports the following most commonly used properties:

- **Height:** the height of the displayed image.
- **Width:** the width of the displayed image.
- **Src:** the source (url) of the image.
- **Alignment-baseline:** the alignment of the image.
- **Scaling:** (Uniform) means the original height width ration ispresderved.

Here is an example of how to load an external graphic using XSL-FO:

```
<fo:block>
          <fo:external-graphic src=”testImage.gif”/>
</fo:block>
```

This will load and display the testImage.gif in the defined block region provided the image is available in the classpath for the FOP transformation.

Another way of embedding images with XSL-FO is to used `<fo:instream-foreign-object>` element. The `<fo:instream-foreign-object>` element is used for inline graphics or other `“generic”` object where the object data resides as descendent of the `<fo:instream-foreign-object>`, typically as an XML element sub tree in a non-XSL namespace. One popular use of `<fo:instream-foreign-object>` is to embed Scalar Vector Graphics (SVG) files. The SVG in lay man's terms is a XML representation of an image. The `<fo:instream-foreign-object>` also supports all the common properties supported by the `<fo:external-graphic>` element. For a detailed description of all the attributes supported by these elements please refer to the W3C XSL-FO website.

Here is an example of loading a SVG file in XSL-FO:

```
<fo:instream-foreign-object>
     <svg:svg xmlns:svg=http://www.w3.org/svg@ width=500px” height=”500px”>
     <xvg:title> An example…</svg:title>
        …….
</svg:svg>
</fo:instream-foreign-object>
```

Thus, there are in total, two ways to display and load images from external file using XSL-FO and FOP. One as an external graphic resource and second as an instream foreign object for images such as SVG. Both these approaches require that the image resource is a file local or remote to the processor. This begs the question, how to load images from a non-file resource. In the next section we will examine how to load images from a non- file resource using XSL-FO.

Loading Images From Non-file resources

As we have seen that loading images from external file resources is fairly easy with XSL-FO. However, recently, I faced a situation where I had to load and display images with FOP when images are stored in a database using Binary Large Objects (BLOB). In this situation, I am not allowed to store the image in any file system. This is a tricky situation and often daunting with limiting prospects. However, there is a clever way to still be able to use these sorts of images. I figured out the following solution architecture for loading and displaying BLOB images using XSL-FO and FOP produce a FOP document.

Figure-1 The BLOB image loading architecture

In this architecture, I used a Data Access Object (DAO) to get the images as a BLOB from the database. A Servlet uses this DAO to get the image data in binary form from the database. The XSL-FO in turn provides the URL of this Servlet as the image source. By this mechanism, the Servlet will serve the image as a binary data stream and XSL –FO will interpret the binary stream as image data and render it as an image.

However straightforward this solution seems, there are few salient points to take note of.

- How do we tell the Servlet, which image to bring back? This means we need a mechanism to pass on to the Servlet some sort of unique identifier for the image. If the image is stored in a database as a BLOB, we can assign a unique id as primary key and pass this primary key value to the Servlet and in turn the DAO to bring back the correct image.
- In order the Servlet to serve the request from XSL-FO, the Servlet must support GET protocol to obtain the data.
- The Servlet when sending the response back must set the content type as part of the response. This helps the XSL-FO to understand the type of image it is dealing with.

Below is an example code for a Servlet to send the image binary stream back to the caller.

```
public class MediaServlet extends HttpServlet {
    protected void doGet(HttpServletRequest req, HttpServletResponse
res) throws ServletException, IOException {
                OutputStream stream = null;
                Attachment attachment = null;

                                String attUrn =
req.getParameter("attachmentURN");
            AttachmentService attachmnetService =
AttachmentServiceFactory.getInstance();
                           try {
                                 stream = res.getOutputStream();
                  attachment = attachmentService.getAttachment(attUrn);

        if(attchment !=null){
             res.setContentType("image/gif");
                res.setContentLength(attachment.getLength());
                    stream.write(attachment.getAttachmentByte());
                  }
             }catch(Exception e){
             }finally{
                    stream.flush();
                    stream.close();
```

```
            }
                }

}
```

Listing 1: The MediaServlet program

In this example, we obtain the image as a binary stream using the AttachmentService Data Access Component. We pass in the attachment URN which is the primary key for the image in the database, and the service returns an Attachment object that contains the binary data for the image. We then write the binary stream back to the caller outputStream.

If we make a call to this service form a XSL-FO stylesheet the MediaServlet will stream the image binary stream back to the XSL-FO stylesheet. XSL-FO will in turn interpret the binary stream by looking at its content-type and content-length and display the image.

The XSL-FO Call

If you are using <fo:external-graphic> element, you can use it in the same way to load a non-file image just by replacing the URN of the image by the url of the Servlet. Here is the example,

```
 <fo:external-graphic src=http://localhhost:8080/servlet/MediaServlet?
attachmentURN=IM293>
</fo:external-graphic>
```

Notice that we are passing the attachment URN as a request parameter. If you need to you can parameterize this URN by passing the data as part of the XML that will be processed. Do not forget, we are doing all these to process an XML data with this XSL-FO and pass through FOP to produce output in a desired format such as PDF etc.

Conclusion

This brings us to the end of this article and this series on FOP and Java. We have in total covered the basics of XSL-FO, converted XML Data to PDF, created print solutions using FOP and finally seen how to use images with FOP from a local file resource and also from non-file resource. This almost covers all the aspects you need to know when using FOP with Java. As FOP uses XSL-FO, you will eventually need to know more about XSL-FO that I have not covered in this series. With this I shall draw an end to this series. Good-bye and Happy Coding.

Samudra Gupta has six years of Java related application development experience. He has been involved in various research based projects in Java including e-commerce based and application design and development projects. He is based in the United Kingdom. In his free time, he is a columnist in different Java Magazines and Journals and loves to play contract bridge.
