Apache FOP: Embedding

How to Embed FOP in a Java application

$Revision: 390222 $

Table of contents

1. Overview

Review Running FOP for important information that applies to embedded applications as well as command-line use, such as options and performance.

To embed Apache FOP in your application, first create a new org.apache.fop.apps.FopFactory instance. This object can be used to launch multiple rendering runs. For each run, create a new org.apache.fop.apps.Fop instance through one of the factory methods of FopFactory. In the method call you specify which output format (i.e. Renderer) to use and, if the selected renderer requires an OutputStream, which OutputStream to use for the results of the rendering. You can customize FOP's behaviour in a rendering run by supplying your own FOUserAgent instance. The FOUserAgent can, for example, be used to set your own Renderer instance (details below). Finally, you retrieve a SAX DefaultHandler instance from the Fop object and use that as the SAXResult of your transformation.

Note:

We recently changed FOP's outer API to what we consider the final API. This might require some changes in your application. The main reasons for these changes were performance improvements due to better reuse of reusable objects and reduced use of static variables for added flexibility in complex environments.

2. Basic Usage Pattern

Apache FOP relies heavily on JAXP. It uses SAX events exclusively to receive the XSL-FO input document. It is therefore a good idea that you know a few things about JAXP (which is a good skill anyway). Let's look at the basic usage pattern for FOP...

Here is the basic pattern to render an XSL-FO file to PDF:

```
import org.apache.fop.apps.FopFactory;
import org.apache.fop.apps.Fop;
import org.apache.fop.apps.MimeConstants;

/*..*/

// Step 1: Construct a FopFactory
// (reuse if you plan to render multiple documents!)
FopFactory fopFactory = FopFactory.newInstance();

// Step 2: Set up output stream.
// Note: Using BufferedOutputStream for performance reasons (helpful with
FileOutputStreams).
OutputStream out = new BufferedOutputStream(new FileOutputStream(new
File("C:/Temp/myfile.pdf")));
```

```
try {
  // Step 3: Construct fop with desired output format
  Fop fop = fopFactory.newFop(MimeConstants.MIME_PDF, out);

  // Step 4: Setup JAXP using identity transformer
  TransformerFactory factory = TransformerFactory.newInstance();
  Transformer transformer = factory.newTransformer(); // identity transformer

  // Step 5: Setup input and output for XSLT transformation
  // Setup input stream
  Source src = new StreamSource(new File("C:/Temp/myfile.fo"));

  // Resulting SAX events (the generated FO) must be piped through to FOP
  Result res = new SAXResult(fop.getDefaultHandler());

  // Step 6: Start XSLT transformation and FOP processing
  transformer.transform(src, res);

} finally {
  //Clean-up
  out.close();
}
```

Let's discuss these 5 steps in detail:

- **Step 1:** You create a new FopFactory instance. The FopFactory instance holds references to configuration information and cached data. It's important to reuse this instance if you plan to render multiple documents during a JVM's lifetime.
- **Step 2:** You set up an OutputStream that the generated document will be written to. It's a good idea to buffer the OutputStream as demonstrated to improve performance.
- **Step 3:** You create a new Fop instance through one of the factory methods on the FopFactory. You tell the FopFactory what your desired output format is. This is done by using the MIME type of the desired output format (ex. "application/pdf"). You can use one of the MimeConstants.* constants. The second parameter is the OutputStream you've setup up in step 2.
- **Step 4** We recommend that you use JAXP Transformers even if you don't do XSLT transformations to generate the XSL-FO file. This way you can always use the same basic pattern. The example here sets up an "identity transformer" which just passes the input (Source) unchanged to the output (Result). You don't have to work with a SAXParser if you don't do any XSLT transformations.
- **Step 5:** Here you set up the input and output for the XSLT transformation. The Source object is set up to load the "myfile.fo" file. The Result is set up so the output of the XSLT transformation is sent to FOP. The FO file is sent to FOP in the form of SAX events which is the most efficient way. Please always avoid saving intermediate results to a file or a memory buffer because that affects performance negatively.
- **Step 6:** Finally, we start the XSLT transformation by starting the JAXP Transformer. As soon as the JAXP Transformer starts to send its output to FOP, FOP itself starts its

processing in the background. When the `transform()` method returns FOP will also have finished converting the FO file to a PDF file and you can close the OutputStream.

Tip!

It's a good idea to enclose the whole conversion in a try..finally statement. If you close the OutputStream in the finally section, this will make sure that the OutputStream is properly closed even if an exception occurs during the conversion.

If you're not totally familiar with JAXP Transformers, please have a look at the Embedding examples below. The section contains examples for all sorts of use cases. If you look at all of them in turn you should be able to see the patterns in use and the flexibility this approach offers without adding too much complexity.

This may look complicated at first, but it's really just the combination of an XSL transformation and a FOP run. It's also easy to comment out the FOP part for debugging purposes, for example when you're tracking down a bug in your stylesheet. You can easily write the XSL-FO output from the XSL transformation to a file to check if that part generates the expected output. An example for that can be found in the Embedding examples (See "ExampleXML2FO").

2.1. Logging

Logging is now a little different than it was in FOP 0.20.5. We've switched from Avalon Logging to Jakarta Commons Logging. While with Avalon Logging the loggers were directly given to FOP, FOP now retrieves its logger(s) through a statically available LogFactory. This is similar to the general pattern that you use when you work with Apache Log4J directly, for example. We call this "static logging" (Commons Logging, Log4J) as opposed to "instance logging" (Avalon Logging). This has a consequence: You can't give FOP a logger for each processing run anymore. The log output of multiple, simultaneously running FOP instances is sent to the same logger.

Note:

We know this may be an issue in multi-threaded server environments if you'd like to know what's going on in every single FOP processing run. We're planning to add an additional feedback facility to FOP which can be used to obtain all sorts of specific feedback (validation messages, layout problems etc.). "Static logging" is mainly interesting for a developer working on FOP and for advanced users who are debugging FOP. We don't consider the logging output to be useful to normal FOP users. Please have some patience until we can add this feature or jump in and help us build it. We've set up a Wiki page which documents what we're going to build.

By default, Jakarta Commons Logging uses JDK logging (available in JDKs 1.4 or higher) as its backend. You can configure Commons Logging to use an alternative backend, for example Log4J. Please consult the documentation for Jakarta Commons Logging on how to configure alternative backends.

2.2. Processing XSL-FO

Once the Fop instance is set up, call `getDefaultHandler()` to obtain a SAX DefaultHandler instance to which you can send the SAX events making up the XSL-FO document you'd like to render. FOP processing starts as soon as the DefaultHandler's `startDocument()` method is called. Processing stops again when the DefaultHandler's `endDocument()` method is called. Please refer to the basic usage pattern shown above to render a simple XSL-FO document.

2.3. Processing XSL-FO generated from XML+XSLT

If you want to process XSL-FO generated from XML using XSLT we recommend again using standard JAXP to do the XSLT part and piping the generated SAX events directly through to FOP. The only thing you'd change to do that on the basic usage pattern above is to set up the Transformer differently:

```
  //without XSLT:
  //Transformer transformer = factory.newTransformer(); // identity
transformer

  //with XSLT:
  Source xslt = new StreamSource(new File("mystylesheet.xsl"));
  Transformer transformer = factory.newTransformer(xslt);
```

3. Input Sources

The input XSL-FO document is always received by FOP as a SAX stream (see the Parsing Design Document for the rationale).

However, you may not always have your input document available as a SAX stream. But with JAXP it's easy to convert different input sources to a SAX stream so you can pipe it into FOP. That sounds more difficult than it is. You simply have to set up the right Source instance as input for the JAXP transformation. A few examples:

- **URL:** `Source src = new StreamSource("http://localhost:8080/testfile.xml");`
- **File:** `Source src = new StreamSource(new File("C:/Temp/myinputfile.xml"));`
- **String:** `Source src = new StreamSource(new StringReader(myString)); // myString is a String`
- **InputStream:** `Source src = new StreamSource(new`

```
MyInputStream(something));
```

- **Byte Array:** `Source src = new StreamSource(new ByteArrayInputStream(myBuffer)); // myBuffer is a byte[] here`
- **DOM:** `Source src = new DOMSource(myDocument); // myDocument is a Document or a Node`
- **Java Objects:** Please have a look at the Embedding examples which contain an example for this.

There are a variety of upstream data manipulations possible. For example, you may have a DOM and an XSL stylesheet; or you may want to set variables in the stylesheet. Interface documentation and some cookbook solutions to these situations are provided in Xalan Basic Usage Patterns.

4. Configuring Apache FOP Programmatically

Apache FOP provides two levels on which you can customize FOP's behaviour: the FopFactory and the user agent.

4.1. Customizing the FopFactory

The FopFactory holds configuration data and references to objects which are reusable over multiple rendering runs. It's important to instantiate it only once (except in special environments) and reuse it every time to create new FOUserAgent and Fop instances.

You can set all sorts of things on the FopFactory:

- The **font base URL** to use when resolving relative URLs for fonts. Example:
  ```
  fopFactory.setFontBaseURL("file:///C:/Temp/fonts");
  ```
- Disable **strict validation**. When disabled FOP is less strict about the rules established by the XSL-FO specification. Example:
  ```
  fopFactory.setStrictValidation(false);
  ```
- Enable an **alternative set of rules for text indents** that tries to mimic the behaviour of many commercial FO implementations, that chose to break the specification in this respect. The default of this option is 'false', which causes Apache FOP to behave exactly as described in the specification. To enable the alternative behaviour, call:
  ```
  fopFactory.setBreakIndentInheritanceOnReferenceAreaBoundary(true);
  ```
- Set the **source resolution** for the document. This is used internally to determine the pixel size for SVG images and bitmap images without resolution information. Default: 72 dpi. Example:
  ```
  fopFactory.setSourceResolution(96); // =96dpi (dots/pixels per Inch)
  ```

- Manually add an **ElementMapping instance**. If you want to supply a special FOP extension you can give the instance to the FOUserAgent. Normally, the FOP extensions can be automatically detected (see the documentation on extension for more info). Example:

```
fopFactory.addElementMapping(myElementMapping); // myElementMapping is a
org.apache.fop.fo.ElementMapping
```

- Set a **URIResolver** for custom URI resolution. By supplying a JAXP URIResolver you can add custom URI resolution functionality to FOP. For example, you can use Apache XML Commons Resolver to make use of XCatalogs. Example:

```
fopFactory.setURIResolver(myResolver); // myResolver is a
javax.xml.transform.URIResolver
```

Note:

Both the FopFactory and the FOUserAgent have a method to set a URIResolver. The URIResolver on the FopFactory is primarily used to resolve URIs on factory-level (hyphenation patterns, for example) and it is always used if no other URIResolver (for example on the FOUserAgent) resolved the URI first.

4.2. Customizing the User Agent

The user agent is the entity that allows you to interact with a single rendering run, i.e. the processing of a single document. If you wish to customize the user agent's behaviour, the first step is to create your own instance of FOUserAgent using the appropriate factory method on FopFactory and pass that to the factory method that will create a new Fop instance:

```
  FopFactory fopFactory = FopFactory.newInstance(); // Reuse the FopFactory if
possible!
  // do the following for each new rendering run
  FOUserAgent userAgent = fopFactory.newFOUserAgent();
  // customize userAgent
  Fop fop = fopFactory.newFop(MimeConstants.MIME_POSTSCRIPT, userAgent, out);
```

You can do all sorts of things on the user agent:

- The **base URL** to use when resolving relative URLs. Example:

```
userAgent.setBaseURL("file:///C:/Temp/");
```

- Set the **producer** of the document. This is metadata information that can be used for certain output formats such as PDF. The default producer is "Apache FOP". Example:

```
userAgent.setProducer("MyKillerApplication");
```

- Set the **creating user** of the document. This is metadata information that can be used for certain output formats such as PDF. Example:

```
userAgent.setCreator("John Doe");
```

- Set the **author** of the document. This is metadata information that can be used for certain output formats such as PDF. Example:

```
userAgent.setAuthor("John Doe");
```

- Override the **creation date and time** of the document. This is metadata information that can be used for certain output formats such as PDF. Example:

```
userAgent.setCreationDate(new Date());
```

- Set the **title** of the document. This is metadata information that can be used for certain output formats such as PDF. Example:

```
userAgent.setTitle("Invoice No 138716847");
```

- Set the **keywords** of the document. This is metadata information that can be used for certain output formats such as PDF. Example:

```
userAgent.setKeywords("XML XSL-FO");
```

- Set the **target resolution** for the document. This is used to specify the output resolution for bitmap images generated by bitmap renderers (such as the TIFF renderer) and by bitmaps generated by Apache Batik for filter effects and such. Default: 72 dpi. Example:

```
userAgent.setTargetResolution(300); // =300dpi (dots/pixels per Inch)
```

- Set **your own Renderer instance**. If you want to supply your own renderer or configure a Renderer in a special way you can give the instance to the FOUserAgent. Normally, the Renderer instance is created by FOP. Example:

```
userAgent.setRendererOverride(myRenderer); // myRenderer is an
org.apache.fop.render.Renderer
```

- Set **your own FOEventHandler instance**. If you want to supply your own FOEventHandler or configure an FOEventHandler subclass in a special way you can give the instance to the FOUserAgent. Normally, the FOEventHandler instance is created by FOP. Example:

```
userAgent.setFOEventHandlerOverride(myFOEventHandler); // myFOEventHandler
is an org.apache.fop.fo.FOEventHandler
```

- Set a **URIResolver** for custom URI resolution. By supplying a JAXP URIResolver you can add custom URI resolution functionality to FOP. For example, you can use Apache XML Commons Resolver to make use of XCatalogs. Example:

```
userAgent.setURIResolver(myResolver); // myResolver is a
javax.xml.transform.URIResolver
```

Note:

Both the FopFactory and the FOUserAgent have a method to set a URIResolver. The URIResolver on the FOUserAgent is used for resolving URIs which are document-related. If it's not set or cannot resolve a URI, the URIResolver from the FopFactory is used.

Note:

You should not reuse an FOUserAgent instance between FOP rendering runs although you can. Especially in multi-threaded environment, this is a bad idea.

5. Using a Configuration File

Instead of setting the parameters manually in code as shown above you can also set many values from an XML configuration file:

```
import org.apache.avalon.framework.configuration.Configuration;
import org.apache.avalon.framework.configuration.DefaultConfigurationBuilder;

/*..*/

DefaultConfigurationBuilder cfgBuilder = new DefaultConfigurationBuilder();
Configuration cfg = cfgBuilder.buildFromFile(new File("C:/Temp/mycfg.xml"));
fopFactory.setUserConfig(cfg);

/* ..or.. */

fopFactory.setUserConfig(new File("C:/Temp/mycfg.xml"));
```

The layout of the configuration file is described on the Configuration page.

6. Hints

6.1. Object reuse

Fop instances shouldn't (and can't) be reused. Please recreate Fop and FOUserAgent instances for each rendering run using the FopFactory. This is a cheap operation as all reusable information is held in the FopFactory. That's why it's so important to reuse the FopFactory instance.

6.2. AWT issues

If your XSL-FO files contain SVG then Apache Batik will be used. When Batik is initialised it uses certain classes in `java.awt` that intialise the Java AWT classes. This means that a daemon thread is created by the JVM and on Unix it will need to connect to a DISPLAY.

The thread means that the Java application may not automatically quit when finished, you will need to call `System.exit()`. These issues should be fixed in the JDK 1.4.

If you run into trouble running FOP on a head-less server, please see the notes on Batik.

6.3. Getting information on the rendering process

To get the number of pages that were rendered by FOP you can call `Fop.getResults()`. This returns a `FormattingResults` object where you can look up the number of pages

produced. It also gives you the page-sequences that were produced along with their id attribute and their numbers of pages. This is particularly useful if you render multiple documents (each enclosed by a page-sequence) and have to know the number of pages of each document.

7. Improving performance

There are several options to consider:

- Whenever possible, try to use SAX to couple the individual components involved (parser, XSL transformer, SQL datasource etc.).
- Depending on the target OutputStream (in case of a FileOutputStream, but not for a ByteArrayOutputStream, for example) it may improve performance considerably if you buffer the OutputStream using a BufferedOutputStream: `out = new java.io.BufferedOutputStream(out);`
  Make sure you properly close the OutputStream when FOP is finished.
- Cache the stylesheet. If you use the same stylesheet multiple times you can set up a JAXP `Templates` object and reuse it each time you do the XSL transformation. (More information can be found here.)
- Use an XSLT compiler like XSLTC that comes with Xalan-J.
- Fine-tune your stylesheet to make the XSLT process more efficient and to create XSL-FO that can be processed by FOP more efficiently. Less is more: Try to make use of property inheritance where possible.

8. Multithreading FOP

Apache FOP may currently not be completely thread safe. The code has not been fully tested for multi-threading issues, yet. If you encounter any suspicious behaviour, please notify us.

There is also a known issue with fonts being jumbled between threads when using the Java2D/AWT renderer (which is used by the -awt and -print output options). In general, you cannot safely run multiple threads through the AWT renderer.

9. Examples

The directory "{fop-dir}/examples/embedding" contains several working examples.

9.1. ExampleFO2PDF.java

This example demonstrates the basic usage pattern to transform an XSL-FO file to PDF using

FOP.

Example XSL-FO to PDF

9.2. ExampleXML2FO.java

This example has nothing to do with FOP. It is there to show you how an XML file can be converted to XSL-FO using XSLT. The JAXP API is used to do the transformation. Make sure you've got a JAXP-compliant XSLT processor in your classpath (ex. Xalan).

Example XML to XSL-FO

9.3. ExampleXML2PDF.java

This example demonstrates how you can convert an arbitrary XML file to PDF using XSLT and XSL-FO/FOP. It is a combination of the first two examples above. The example uses JAXP to transform the XML file to XSL-FO and FOP to transform the XSL-FO to PDF.

Example XML to PDF (via XSL-FO)

The output (XSL-FO) from the XSL transformation is piped through to FOP using SAX events. This is the most efficient way to do this because the intermediate result doesn't have to be saved somewhere. Often, novice users save the intermediate result in a file, a byte array or a DOM tree. We strongly discourage you to do this if it isn't absolutely necessary. The performance is significantly higher with SAX.

9.4. ExampleObj2XML.java

This example is a preparatory example for the next one. It's an example that shows how an arbitrary Java object can be converted to XML. It's an often needed task to do this. Often people create a DOM tree from a Java object and use that. This is pretty straightforward. The example here, however, shows how to do this using SAX, which will probably be faster and not even more complicated once you know how this works.

Example Java object to XML

For this example we've created two classes: ProjectTeam and ProjectMember (found in xml-fop/examples/embedding/java/embedding/model). They represent the same data structure found in xml-fop/examples/embedding/xml/xml/projectteam.xml. We want to serialize to XML a project team with several members which exist as Java objects. Therefore we created the two classes: ProjectTeamInputSource and ProjectTeamXMLReader (in the same place as ProjectTeam above).

The XMLReader implementation (regard it as a special kind of XML parser) is responsible for creating SAX events from the Java object. The InputSource class is only used to hold the

ProjectTeam object to be used.

Have a look at the source of ExampleObj2XML.java to find out how this is used. For more detailed information see other resources on JAXP (ex. An older JAXP tutorial).

9.5. ExampleObj2PDF.java

This example combines the previous and the third to demonstrate how you can transform a Java object to a PDF directly in one smooth run by generating SAX events from the Java object that get fed to an XSL transformation. The result of the transformation is then converted to PDF using FOP as before.

Example Java object to PDF (via XML and XSL-FO)

9.6. ExampleDOM2PDF.java

This example has FOP use a DOMSource instead of a StreamSource in order to use a DOM tree as input for an XSL transformation.

9.7. ExampleSVG2PDF.java (PDF Transcoder example)

This example shows the usage of the PDF Transcoder, a sub-application within FOP. It is used to generate a PDF document from an SVG file.

9.8. Final notes

These examples should give you an idea of what's possible. It should be easy to adjust these examples to your needs. Also, if you have other examples that you think should be added here, please let us know via either the fop-users or fop-dev mailing lists. Finally, for more help please send your questions to the fop-users mailing list.