Image Loader Framework

Table of contents

1 Overview

Apache XML Graphics Commons contains a unified framework for loading and processing images (bitmap and vector). The package name is `org.apache.xmlgraphics.image.loader`. Key features:

- Unified basic API for all supported image types.
- Image "Preloading": It allows automatic detection on the image content type and can extract the intrinsic size (in pixels and length units) of the image without loading the whole image into memory in most cases. Apache FOP uses this as it only needs the size of the image to do the layout. The image is only fully read at the rendering stage.
- Image conversion facility: Images can be converted into different representations depending on the needs of the consumer.
- Supported formats: All bitmap formats for which there are codecs for the ImageIO API (like JPEG, PNG, GIF etc.), EPS, EMF. These formats are bundled. Other formats such as SVG and WMF are available through plug-ins hosted elsewhere.
- Supported in-memory representations:
  - RenderedImage/BufferedImage
  - raw/undecoded (JPEG, EPS, CCITT group 3/4)
  - Java2D (Images painted through Graphics2D)
  - XML DOM (for SVG, MathML etc.)
  - Additional representations can be added as necessary.
- Custom image loaders and converters can be dynamically plugged in. Automatic plug-in detection through the application classpath.
- An image cache speeds up the processing for images that are requested multiple times.

2 Tutorial

2.1 Setting up the manager

Before we can start to work with the package we need to set up the `ImageManager`. It provides convenience methods to load and convert images and holds the image cache.

The `ImageManager` needs an `ImageContext`. This interface provides the `ImageManager` with important context and configuration data. Currently this is only the source resolution. The `ImageManager` and `ImageContext` are intended to be shared within an application.

```
import org.apache.xmlgraphics.image.loader.ImageContext;
import org.apache.xmlgraphics.image.loader.ImageManager;
import org.apache.xmlgraphics.image.loader.impl.DefaultImageContext;

[..]

  ImageManager imageManager = new ImageManager(new DefaultImageContext());
```

> **Note:**
> In this example, `DefaultImageContext` is used. You may need to write your own implementation of `ImageContext` for your use case.

2.2 Preloading an image

In order to load an image, it needs to be "preloaded" first, i.e. the image content type is detected and the intrinsic size of the image is determined. The result of this process is an `ImageInfo` instance which contains the URI, MIME type and intrinsic size. In most cases, this is done without loading the whole image (see SPI section below for information on exceptions to this rule).

Preloading is normally done through the `ImageManager`'s `getImageInfo()` method. For this operation an `ImageSessionContext` needs to be provided. It is responsible for supplying JAXP `Source` objects, URI resolution and providing other information needed for the image operations. In simple cases you can simply use `DefaultImageSessionContext`, but often you will want to write your own implementation of `ImageSessionContext`. In that case, it's recommended to subclass `AbstractImageSessionContext` which lets you avoid rewriting a lot of code for providing `Source` objects.

```
import org.apache.xmlgraphics.image.loader.ImageInfo;
import org.apache.xmlgraphics.image.loader.ImageSessionContext;
import org.apache.xmlgraphics.image.loader.impl.DefaultImageSessionContext;

[..]
  ImageSessionContext sessionContext = new DefaultImageSessionContext(
          imageManager.getImageContext(), null);

  ImageInfo info = imageManager.getImageInfo(uri, sessionContext);
```

2.3 Loading an image

Once the image is "preloaded", it can be fully loaded in the form/flavor that is needed by the consuming application. The required flavor is indicated through the `ImageFlavor` class. If you want the image as a bitmap image in memory, you could request an `ImageFlavor.RENDERED_IMAGE`. Again, the `ImageSessionContext` will be needed.

```
import org.apache.xmlgraphics.image.loader.Image;
import org.apache.xmlgraphics.image.loader.ImageFlavor;

[..]
  Image img = this.imageManager.getImage(
          info, ImageFlavor.RENDERED_IMAGE, sessionContext);

  ImageRendered imageRend = (ImageRendered)img;
  RenderedImage ri = imageRend.getRenderedImage();
  //...and do anything with the RenderedImage
```

In this example above, we simply acquire the image as a RenderedImage instance. If the original image was a vector graphic image (SVG, WMF etc.), it's automatically converted to a bitmap image. Note: The resolution of the created image is controlled by the target resolution returned by the `ImageSessionContext`.

Of course, the framework can only provide images in the formats, it has image loaders or image converters for. An example: It is possible to load EPS images, but they can only be provided in raw form. In order to provide it as a bitmap image, a PostScript interpreter would be needed to interpret the PostScript code. This

interpreter would be integrated using an `ImageConverter` implementation (see SPI section below). If the requested form of the image cannot be provided you will get an `ImageException` on which you'll have to react as needed.

In Apache FOP, each renderer supports a different set of image flavors that can be embedded in the target format. For example: The PDF renderer can deal with Java2D image, bitmaps, XML, native JPEG and CCITT images. The PCL renderer, however, can only consume bitmap images. So, if you can accept more than one flavor, the package allows you to specify all of them in an ordered list (the first in the list is the preferred format). The package will then try to return the best representation possible. Here's a code example:

```
import org.apache.xmlgraphics.image.loader.Image;
import org.apache.xmlgraphics.image.loader.ImageFlavor;

[..]
  final ImageFlavor[] flavors = new ImageFlavor[]
          {ImageFlavor.GRAPHICS2D,
           ImageFlavor.BUFFERED_IMAGE,
           ImageFlavor.RENDERED_IMAGE};

  Image img = manager.getImage(
          info, flavors, sessionContext);

  if (img instanceof ImageGraphics2D) {
      //handle Java2D/Graphics2D image
  } else if (img instanceof ImageRendered) {
      //handle BufferedImage and RenderedImage
      //(BufferedImage is a subclass of RenderedImage)
  } else {
      throw new IllegalStateException("Unexpected flavor");
  }
```

Note:

While each `BufferedImage` is also a `RenderedImage`, it can be more efficient to also specify `ImageFlavor.BUFFERED_IMAGE` in the flavor array.

3 Tips & Tricks

If you are loading bitmap images and you get an error like `"Cannot load image (no suitable loader/converter combination available) for myimage.tif (image/tiff)`, you maybe be missing the necessary ImageIO codec to decode the image. A number of well-written codecs can be found in JAI Image I/O Tools Project. Just download the distribution and add the JAR to the classpath. ImageIO will automatically pick up the new codecs and they will subsequently be available to the image framework.

4 Service Provider Interface (SPI, Plug-ins)

The whole image framework is designed to be highly extensible. There are various extension points where new functionality can be added. The three main SPI interfaces are:

- `ImagePreloader`: detects the content type and preloads an image
- `ImageLoader` and `ImageLoaderFactory`: loads images

- `ImageConverter`: converts images from one representation into another

If you plan to write an implementation of one of the above interfaces, please also take a look at the existing implementations for reference.

Throughout the SPI, you'll find a `Map` parameter (hints) in the most important methods. That's a way to supply additional information to the implementation by the caller. For example, the source and target resolutions from the image (session) context is stored in the hints. The implementation should not rely on the presence of specialized information and should always have sensible defaults to rely on in this case.

4.1 ImagePreloader

The first task is identifying whether the implementation supports the given image. If the image is loaded using an ImageInputStream it is important to always reset the stream position to the beginning of the file at the end of the `preloadImage()` method, because all registered preloaders are check in turn until one implementation signals that it supports the format. In that case, it has to extract only the minimal information from the image necessary to identify the image's intrinsic size. For most formats, this is doable without loading the whole image into memory.

However, for some formats (like MathML or WMF), loading the whole image at preloading time is hard to avoid since the image's size can only be determined that way. In such a case, the `ImagePreloader` implementations shall pass the loaded document to the respective `ImageLoader` through the custom objects that can be attached to the `ImageInfo` object. If the preloader loads the whole document, it shall close the given `Source` object (calling `ImageUtil.closeQuietly(Source)`).

The priority the implementation reports is used to sort all registered implementations. This is to fine-tune the inner workings and to optimize performance since some formats are usually used more frequently than others.

> **Note:**
> Normally, if you implement an `ImagePreloader` you will also need to implement the respective `ImageLoader/ImageLoaderFactory`, or vice versa.

4.2 ImageLoader and ImageLoaderFactory

The factory interface has been created to allow checking if some library that an implementation depends on is really in the classpath so it can report back that the `ImageLoader` is not funtional. The factory also reports what kind of image formats it supports and which image flavors it can return. There can be a complex relationship between the two. It is recommmended, however, to write smaller implementations rather than big, almighty ones.

The usage penalty is used when constructing image conversion pipelines. There can be multiple ways to provide an image in one of the supported flavors and this value helps to make the best decision.

While the factory basically just provides information and creates new `ImageLoader` instances, the image loaders are doing the actual leg work of decoding the images. The image flavor returned by the loader must match the flavor that is returned by `getTargetFlavor()`.

4.3 ImageConverter

The image converter is responsible to transform one image representation into another. Bundled implementations support these conversions: Java2D to bitmap, bitmap to Java2D and RenderedImage to "raw" PNG. Ideas for additional image converters could be: PDF to Java2D, EPS to Java2D or MathML to SVG or Java2D.

Each ImageConverter comes with a usage penalty which is used when constructing conversion pipelines so the pipeline with the least penalty value can be chosen. This is necessary as the consuming application my support multiple image flavors and there can be multiple ways to convert an image in one of the requested image flavors. Internally, Dijkstra's shortest path algorithm is used to find the best path using the penalties as "way lengths".