OSGiSupport

OSGi Support for Apache XML Graphics Commons

Starting point

All three subprojects (XGC, Batik and FOP) use the "Service Provider" mechanism from the JAR specification for detecting plug-ins (loading files in the META-INF/services directory of various JARs).

OSGi is a popular module system and service platform for the JVM. OSGi doesn't have a hierarchical class loader setup like traditional Java applications which is why a provider consumer may not see all available plug-ins anymore if they are not compiled together into an ugly monster-JAR.

Also, the XG products don't currently provide OSGi metadata in the manifest, yet. Therefore, Batik and FOP cannot currently be run without hacks in an OSGi environment.

Batik and FOP can profit from lifecycle management in an OSGi environment, i.e. new plug-ins can be added/replaced/removed at runtime without restarting the JVM. Currently, there is no prevision for this kind of behaviour but it is easily possible to change the XG products so the static legacy-approach can be mapped to a new approach needed for supporting the full OSGi lifecycle dynamics.

Requirements

Extensions are an important part of XML-based formats like SVG and XSL-FO. Adding OSGi support to XG products should introduce as few new dependencies as possible and make it easy for plug-in developers to provide OSGi-capable versions of their plug-ins. The whole thing needs to be implemented so the same JARs can be run in a classic Java environment (with a hierarchical class loader) as well as in an OSGi-environment. In a classic Java environment, there must not be any runtime-dependencies on OSGi-specific JARs.

Solutions

A possible solution was presented in 2009 but never published (although privately used in production). That solution has some downsides. It adds an external dependency in addition to the OSGi Core API.

The latest OSGi R4 draft offers a simpler solution as part of RFC 167 (SPI Service Loader support). The following section describes how this can be applied to the XG project.

If XG was on Java 6 we could use the `java.util.ServiceLoader` and therefore only require minor modifications to all three products. But that approach would still not profit from the OSGi lifecycle dynamics.

Implementing OSGi RFC 167

More or less the following steps have to be performed to make XG products fully OSGi-compatible while preserving the full functionality in a classic Java environment:

ServiceProviderListener

We can define a ServiceProviderListener interface in XGC with the following signature:

```
package org.apache.xmlgraphics.util.spi;

public interface ServiceProviderListener<T> {
    void providerAdded(T provider);
    void providerRemoved(T provider);
}
```

All parts that are now calling `org.apache.xmlgraphics.util.Service` for detecting plug-ins have to be refactored to directly receive notifications about additions and removals of service providers (service lifecycle). Reusable convenience classes should be easy to implement to make the work easier. Minor complications could arise from the fact the some clients of the `Service` class request class names while other request instances. Also some concurrency issues have to be considered during the refactoring as services can come and go at any time in an OSGi environment.

The only dependency on OSGi (Core) can be restricted to the `org.apache.xmlgraphics.util.spi` package and it will only be an optional runtime dependency.

Generating OSGi metadata

For a normal JAR to become an "OSGi bundle", special entries need to be added to the JAR's manifest. This includes declarations of the JAR's dependent Java packages and the set of packages it "exports". This can be done via the Bnd tool or another tool like the OSGi Bundle Utility for Apache Ant. They automatically calculate the dependencies by bytecode inspection and generate the necessary metadata. This can be integrated into the Ant builds.

Plug-ins will only need add OSGi metadata and the "SPI-Provider: *" manifest entry to their JARs. They don't need to register any OSGi services themselves.

Apache Aries SPI-Fly

The proof-of-concept implementation of RFC167 is SPI-Fly from the Apache Aries project. It can be used to verify if the new approach works.

Plug-in points

Plug-in interface	Potential problems
XML Graphics Commons	
org.apache.xmlgraphics.image.loader.spi.ImagePreloader	
org.apache.xmlgraphics.image.loader.spi.ImageLoaderFactory	
org.apache.xmlgraphics.image.loader.spi.ImageConverter	
org.apache.xmlgraphics.image.writer.ImageWriter	
javax.xml.transform.URIResolver	
FOP	
org.apache.fop.events.model.EventModelFactory	
org.apache.fop.events.EventExceptionManager.ExceptionFactory	
org.apache.fop.fo.ElementMapping	switch from strings to instances
org.apache.fop.render.ImageHandler	
org.apache.fop.render.intermediate.IFDocumentHandler	
org.apache.fop.fo.FOEventHandler	
org.apache.fop.render.Renderer	
org.apache.fop.render.XMLHandler	
org.apache.fop.util.ContentHandlerFactory	
org.apache.fop.util.text.AdvancedMessageFormat.PartFactory	
org.apache.fop.util.text.AdvancedMessageFormat.ObjectFormatter	
org.apache.fop.util.text.AdvancedMessageFormat.Function	
Batik	
org.apache.batik.apps.svgbrowser.SquiggleInputHandler	
org.apache.batik.bridge.BridgeExtension	sorting of providers
org.apache.batik.dom.DomExtension	sorting of providers
org.apache.batik.ext.awt.image.spi.RegistryEntry	
org.apache.batik.ext.awt.image.spi.ImageWriter	to be replaced by XGC
org.apache.batik.script.InterpreterFactory	
org.apache.batik.util.ParsedURLProtocolHandler	

Special cases

Apache FOP contains a mechanism to load fonts from JARs. This will likely also not work under OSGi out of the box. But adding a BundleListener watching out for "font bundles" will be easy to implement.

Status

No definitive timetable exists for implementing the above at this time.