Events/Processing Feedback

$Revision: 932497 $

Table of contents

1. Introduction

In versions until 0.20.5, FOP used Avalon-style Logging where it was possible to supply a logger per processing run. During the redesign the logging infrastructure was switched over to Commons Logging which is (like Log4J or java.util.logging) a "static" logging framework (the logger is accessed through static variables). This made it very difficult in a multi-threaded system to retrieve information for a single processing run.

With FOP's event subsystem, we'd like to close this gap again and even go further. The first point is to realize that we have two kinds of "logging". Firstly, we have the logging infrastructure for the (FOP) developer who needs to be able to enable finer log messages for certain parts of FOP to track down a certain problem. Secondly, we have the user who would like to be informed about missing images, overflowing lines or substituted fonts. These messages (or events) are targeted at less technical people and may ideally be localized (translated). Furthermore, tool and solution builders would like to integrate FOP into their own solutions. For example, an FO editor should be able to point the user to the right place where a particular problem occurred while developing a document template. Finally, some integrators would like to abort processing if a resource (an image or a font) has not been found, while others would simply continue. The event system allows to react on these events.

On this page, we won't discuss logging as such. We will show how the event subsystem can be used for various tasks. We'll first look at the event subsystem from the consumer side. Finally, the production of events inside FOP will be discussed (this is mostly interesting for FOP developers only).

2. The consumer side

The event subsystem is located in the `org.apache.fop.events` package and its base is the `Event` class. An instance is created for each event and is sent to a set of `EventListener` instances by the `EventBroadcaster`. An `Event` contains:

- an event ID,
- a source object (which generated the event),
- a severity level (Info, Warning, Error and Fatal Error) and
- a map of named parameters.

The `EventFormatter` class can be used to translate the events into human-readable, localized messages.

A full example of what is shown here can be found in the `examples/embedding/java/embedding/events` directory in the FOP distribution. The example can also be accessed via the web.

2.1. Writing an EventListener

The following code sample shows a very simple EventListener. It basically just sends all events to System.out (stdout) or System.err (stderr) depending on the event severity.

```
import org.apache.fop.events.Event;
import org.apache.fop.events.EventFormatter;
import org.apache.fop.events.EventListener;
import org.apache.fop.events.model.EventSeverity;

/** A simple event listener that writes the events to stdout and stderr. */
public class SysOutEventListener implements EventListener {

    /** {@inheritDoc} */
    public void processEvent(Event event) {
        String msg = EventFormatter.format(event);
        EventSeverity severity = event.getSeverity();
        if (severity == EventSeverity.INFO) {
            System.out.println("[INFO ] " + msg);
        } else if (severity == EventSeverity.WARN) {
            System.out.println("[WARN ] " + msg);
        } else if (severity == EventSeverity.ERROR) {
            System.err.println("[ERROR] " + msg);
        } else if (severity == EventSeverity.FATAL) {
            System.err.println("[FATAL] " + msg);
        } else {
            assert false;
        }
    }
}
```

You can see that for every event the method `processEvent` of the `EventListener` will be called. Inside this method you can do whatever processing you would like including throwing a `RuntimeException`, if you want to abort the current processing run.

The code above also shows how you can turn an event into a human-readable, localized message that can be presented to a user. The `EventFormatter` class does this for you. It provides additional methods if you'd like to explicitly specify the locale.

It is possible to gather all events for a whole processing run so they can be evaluated afterwards. However, care should be taken about memory consumption since the events provide references to objects inside FOP which may themselves have references to other objects. So holding on to these objects may mean that whole object trees cannot be released!

2.2. Adding an EventListener

To register the event listener with FOP, get the `EventBroadcaster` which is associated with the user agent (`FOUserAgent`) and add it there:

```
FOUserAgent foUserAgent = fopFactory.newFOUserAgent();
foUserAgent.getEventBroadcaster().addEventListener(new SysOutEventListener());
```

Please note that this is done separately for each processing run, i.e. for each new user agent.

2.3. An additional listener example

Here's an additional example of an event listener:

By default, FOP continues processing even if an image wasn't found. If you have more strict requirements and want FOP to stop if an image is not available, you can do something like the following

in the simplest case:

```
public class MyEventListener implements EventListener {

    public void processEvent(Event event) {
        if ("org.apache.fop.ResourceEventProducer".equals(
                event.getEventGroupID())) {
            event.setSeverity(EventSeverity.FATAL);
        } else {
            //ignore all other events (or do something of your choice)
        }
    }

}
```

Increasing the event severity to FATAL will signal the event broadcaster to throw an exception and stop further processing. In the above case, all resource-related events will cause FOP to stop processing.

You can also customize the exception to throw (you can may throw a RuntimeException or subclass yourself) and/or which event to respond to:

```
public class MyEventListener implements EventListener {

    public void processEvent(Event event) {
        if ("org.apache.fop.ResourceEventProducer.imageNotFound"
                .equals(event.getEventID())) {

            //Get the FileNotFoundException that's part of the event's parameters
            FileNotFoundException fnfe =
(FileNotFoundException)event.getParam("fnfe");

            throw new RuntimeException(EventFormatter.format(event), fnfe);
        } else {
            //ignore all other events (or do something of your choice)
        }
    }

}
```

This throws a `RuntimeException` with the `FileNotFoundException` as the cause. Further processing effectively stops in FOP. You can catch the exception in your code and react as you see necessary.

3. The producer side (for FOP developers)

This section is primarily for FOP and FOP plug-in developers. It describes how to use the event subsystem for producing events.

Note:
The event package has been designed in order to be theoretically useful for use cases outside FOP. If you think this is interesting independently from FOP, please talk to us.

3.1. Producing and sending an event

The basics are very simple. Just instantiate an `Event` object and fill it with the necessary parameters.

Then pass it to the `EventBroadcaster` which distributes the events to the interested listeneners. Here's a code example:

```
Event ev = new Event(this, "complain", EventSeverity.WARN,
        Event.paramsBuilder()
            .param("reason", "I'm tired")
            .param("blah", new Integer(23))
            .build());
EventBroadcaster broadcaster = [get it from somewhere];
broadcaster.broadcastEvent(ev);
```

The `Event.paramsBuilder()` is a fluent interface to help with the build-up of the parameters. You could just as well instantiate a `Map` (`Map<String, Object>`) and fill it with values.

3.2. The EventProducer interface

To simplify event production, the event subsystem provides the `EventProducer` interface. You can create interfaces which extend `EventProducer`. These interfaces will contain one method per event to be generated. By contract, each event method must have as its first parameter a parameter named "source" (Type Object) which indicates the object that generated the event. After that come an arbitrary number of parameters of any type as needed by the event.

The event producer interface does not need to have any implementation. The implementation is produced at runtime by a dynamic proxy created by `DefaultEventBroadcaster`. The dynamic proxy creates `Event` instances for each method call against the event producer interface. Each parameter (except "source") is added to the event's parameter map.

To simplify the code needed to get an instance of the event producer interface it is suggested to create a public inner provider class inside the interface.

Here's an example of such an event producer interface:

```
public interface MyEventProducer extends EventProducer {

    public class Provider {

        public static MyEventProducer get(EventBroadcaster broadcaster) {
            return
(MyEventProducer)broadcaster.getEventProducerFor(MyEventProducer.class);
        }
    }

    /**
     * Complain about something.
     * @param source the event source
     * @param reason the reason for the complaint
     * @param blah the complaint
     * @event.severity WARN
     */
    void complain(Object source, String reason, int blah);

}
```

To produce the same event as in the first example above, you'd use the following code:

```
EventBroadcaster broadcaster = [get it from somewhere];
TestEventProducer producer = TestEventProducer.Provider.get(broadcaster);
producer.complain(this, "I'm tired", 23);
```

3.3. The event model

Inside an invocation handler for a dynamic proxy, there's no information about the names of each parameter. The JVM doesn't provide it. The only thing you know is the interface and method name. In order to properly fill the `Event`'s parameter map we need to know the parameter names. These are retrieved from an event object model. This is found in the `org.apache.fop.events.model` package. The data for the object model is retrieved from an XML representation of the event model that is loaded as a resource. The XML representation is generated using an Ant task at build time (`ant resourcegen`). The Ant task (found in `src/codegen/java/org/apache/fop/tools/EventProducerCollectorTask.java`) scans FOP's sources for descendants of the `EventProducer` interface and uses QDox to parse these interfaces.

The event model XML files are generated during build by the Ant task mentioned above when running the "resourcegen" task. So just run `"ant resourcegen"` if you receive a `MissingResourceException` at runtime indicating that `"event-model.xml"` is missing.

Primarily, the QDox-based collector task records the parameters' names and types. Furthermore, it extracts additional attributes embedded as Javadoc comments from the methods. At the moment, the only such attribute is "@event.severity" which indicates the default event severity (which can be changed by event listeners). The example event producer above shows the Javadocs for an event method.

There's one more information that is extracted from the event producer information for the event model: an optional primary exception. The first exception in the "throws" declaration of an event method is noted. It is used to throw an exception from the invocation handler if the event has an event severity of "FATAL" when all listeners have been called (listeners can update the event severity). Please note that an implementation of `org.apache.fop.events.EventExceptionManager$ExceptionFactory` has to be registered for the `EventExceptionManager` to be able to construct the exception from an event.

For a given application, there can be multiple event models active at the same time. In FOP, each renderer is considered to be a plug-in and provides its own specific event model. The individual event models are provided through an `EventModelFactory`. This interface is implemented for each event model and registered through the service provider mechanism (see the plug-ins section for details).

3.4. Event severity

Four different levels of severity for events has been defined:

1. INFO: informational only
2. WARN: a Warning
3. ERROR: an error condition from which FOP can recover. FOP will continue processing.
4. FATAL: a fatal error which causes an exception in the end and FOP will stop processing.

Event listeners can choose to ignore certain events based on their event severity. Please note that you may recieve an event "twice" in a specific case: if there is a fatal error an event is generated and sent to

the listeners. After that an exception is thrown with the same information and processing stops. If the fatal event is shown to the user and the following exception is equally presented to the user it may appear that the event is duplicated. Of course, the same information is just published through two different channels.

3.5. Plug-ins to the event subsystem

The event subsystem is extensible. There are a number of extension points:

- **`org.apache.fop.events.model.EventModelFactory`:** Provides an event model to the event subsystem.
- **`org.apache.fop.events.EventExceptionManager$ExceptionFactory`:** Creates exceptions for events, i.e. turns an event into a specific exception.

The names in bold above are used as filenames for the service provider files that are placed in the `META-INF/services` directory. That way, they are automatically detected. This is a mechanism defined by the JAR file specification.

3.6. Localization (L10n)

One goal of the event subsystem was to have localized (translated) event messages. The `EventFormatter` class can be used to convert an event to a human-readable message. Each `EventProducer` can provide its own XML-based translation file. If there is none, a central translation file is used, called "EventFormatter.xml" (found in the same directory as the `EventFormatter` class).

The XML format used by the `EventFormatter` is the same as Apache Cocoon's catalog format. Here's an example:

```
<?xml version="1.0" encoding="UTF-8"?>
<catalogue xml:lang="en">
  <message key="locator">
    [ (See position {loc})| (See {#gatherContextInfo})| (No context info available)]
  </message>
  <message
key="org.apache.fop.render.rtf.RTFEventProducer.explicitTableColumnsRequired">
    RTF output requires that all table-columns for a table are defined. Output will
be incorrect.{{locator}}
  </message>
  <message key="org.apache.fop.render.rtf.RTFEventProducer.ignoredDeferredEvent">
    Ignored deferred event for {node} ({start,if,start,end}).{{locator}}
  </message>
</catalogue>
```

The example (extracted from the RTF handler's event producer) has message templates for two event methods. The class used to do variable replacement in the templates is `org.apache.fop.util.text.AdvancedMessageFormat` which is more powerful than the `MessageFormat` classes provided by the Java class library (`java.util.text` package).

"locator" is a template that is reused by the other message templates by referencing it through "{{locator}}". This is some kind of include command.

Normal event parameters are accessed by name inside single curly braces, for example: "{node}". For

objects, this format just uses the `toString()` method to turn the object into a string, unless there is an `ObjectFormatter` registered for that type (there's an example for `org.xml.sax.Locator`).

The single curly braces pattern supports additional features. For example, it is possible to do this: "{start,if,start,end}". "if" here is a special field modifier that evaluates "start" as a boolean and if that is true returns the text right after the second comma ("start"). Otherwise it returns the text after the third comma ("end"). The "equals" modifier is similar to "if" but it takes as an additional (comma-separated) parameter right after the "equals" modifier, a string that is compared to the value of the variable. An example: {severity,equals,EventSeverity:FATAL,,some text} (this adds "some text" if the severity is not FATAL).

Additional such modifiers can be added by implementing the `AdvancedMessageFormat$Part` and `AdvancedMessageFormat$PartFactory` interfaces.

Square braces can be used to specify optional template sections. The whole section will be omitted if any of the variables used within are unavailable. Pipe (|) characters can be used to specify alternative sub-templates (see "locator" above for an example).

Developers can also register a function (in the above example: `{#gatherContextInfo})` to do more complex information rendering. These functions are implementations of the `AdvancedMessageFormat$Function` interface. Please take care that this is done in a locale-independent way as there is no locale information available, yet.