Creating a Pipeline

by Flavio Tordini

A pipeline is made of a set of connected filters and attributes. Pipelines are defined by means of an XML document conforming to the Pipeline Schema. The XML file must be named `pipeline.xml` in a task directory (example: `$XFP_HOME/tasks/myTask/pipeline.xml`). Filters must be correctly connected, input and output data types must match. If a filter output type implements the `java.util.Iterator` interface, children filters will be executed for each element in the iterator. You may use existing filters or create new ones from scratch.

Here's a simple example. This pipeline gets the XFP homepage via HTTP and stores it on a local file.

```
<pipeline name="httpTest">
  <filter name="httpGet" class="org.xfp.filters.HTTPGet">
    <property name="url">http://xfp.sourceforge.net</property>
    <filter name="fileWriter" class="org.xfp.filters.convert.FileWriter">
      <property name="filename">myfile.html</property>
    </filter>
  </filter>
</pipeline>
```

In this example filter `httpGet` will be invoked with no input and its output will be passed to filter `fileWriter`. For more examples take a look at `$XFP_HOME/samples`.

1. Filters

A filter is made of an input, an output and set of custom properties. A filter can contain 0 or more children filters accepting as input type the output type of the parent filter. Children filters process the output of the parent filter. A filter can support many input data types, but will always output data of the same type.

2. Attributes

An attribute is any stateful object used by filters to accomplish a specific task. For example attributes maybe used to hold the stateful information such as an FTP session or JavaMail session, or an instance of a heavyweight business object. Attributes may be placed at three different scope levels:

- **Global level**. Attributes defined in $XFP_HOME/conf/engine.xml are global to all the pipelines. They are initialized at initialization time and disposed when XFP is stopped.
- **Pipeline level**. These attributes have a pipeline scope, they are only visible to the filters of a specific pipeline. They are initialized each time a pipeline is executed and disposed when the pipeline completes.
- **Filter level**. These attributes are visible only to a single filter. They are initialized just before filter a execution and disposed just after the filter produced its output.

Here's an example of attribute usage:

```
<attribute name="ftpClient" class="org.xfp.components.FTPClientComponent">
  <property name="server">&ftp.host;</property>
  <property name="userid">&ftp.user;</property>
  <property name="password">&ftp.password;</property>
  <property name="remotedir">&ftp.remotedir;</property>
</attribute>
```

3. Setting properties

Properties are used to parameterize the behaviour of a filter. Use the `property` tag to set filter properties. XFP uses internally the Jakarta Commons BeanUtils package to pass the property values to filters.

The general set of possible property types supported by a JavaBean can be broken into three categories -- some of which are supported by the standard JavaBeans specification, and some of which are uniquely supported by the *BeanUtils* package:

- **Simple** - Simple, or scalar, properties have a single value that may be retrieved or modified. The underlying property type might be a Java language primitive (such as `int`, a simple object (such as a `java.lang.String`), or a more complex object whose class is defined either by the Java language, by the application, or by a class library included with the application.
  ```
  <property name="myProperty">myValue</property>
  ```
- **Indexed** - An indexed property stores an ordered collection of objects (all of the same type) that can be individually accessed by an integer-valued, non-negative index (or subscript). Alternatively, the entire set of values may be set or retrieved using an array. As an extension to the JavaBeans specification, the *BeanUtils* package considers any property whose underlying data type is `java.util.List` (or an implementation of List) to be indexed as well.
  ```
  <property name="myProperty[index]">myValue</property>
  ```
- **Mapped** - As an extension to standard JavaBeans APIs, the *BeanUtils* package considers any property whose underlying value is a `java.util.Map` to be "mapped". You can set and retrieve individual values via a String-valued key.
  ```
  <property name="myProperty(key)">myValue</property>
  ```

See BeanUtils Documentation for more info.

4. ${} expressions

Ant-style ${} expressions can be used in the value of filter properties. This is done by placing the variable name between "${" and "}" in the property value. The expression can be the name of a processed filter, the name of an attribute or the name of a system property. For example, if there is a "dir" variable with the value "mydir", then this could be used in an property like this: ${dir}/classes. This is resolved at run-time as mydir/classes.

```
<property name="myProperty">${dir}/classes</property>
```

There are some default variables bound the filter execution context:

name	value
xfp.task.home	Home directory of the task.
null	A null pointer.

5. Overriding filter input

By default the filter input is the output of the parent filter. This can be overridden by using the `input` attribute.

```
<filter name="myFilter" class="org.myFilter" input="${otherfilter}">
```

6. Overriding filter output type

Some general purpose filters may output generic data types such as `java.lang.Object`. To cast the output type to the actual data type use the `outputType` attribute.

```
<filter name="myFilter" class="org.MyFilter" outputType="java.lang.String">
```

7. Error handling

In XFP you cannot really handle errors, but you can specify how the pipeline execution is affected by an exception on a filter basis. The `filter` tag supports the `onError` attribute. There are two fixed values for this attribute:

value	description
exit	This is the default value. If an exception occurs pipeline execution is halted.
break	If an exception occurs breaks branch execution by skipping all nested filters and the pipeline execution jumps to the next filter branch (i.e. to the next sibling filter, if any).

```
<filter name="myFilter" class="org.MyFilter" onError="break">
```

But what if an exception occurs during an iteration? Should we break the loop or continue with the next element? There's another attribute in the `filter` tag: `onLoopError`. There are two fixed values for this attribute:

value	description
continue	This is the default value. If an exception occurs continue with the next iteration (similar to the java `continue` keyword).
break	If an exception occurs breaks the loop (similar to the java `break` keyword).

```
<filter name="myFilter" class="org.MyFilter" onLoopError="break">
```

Note:

Note that `onLoopError` does not override `onError`.