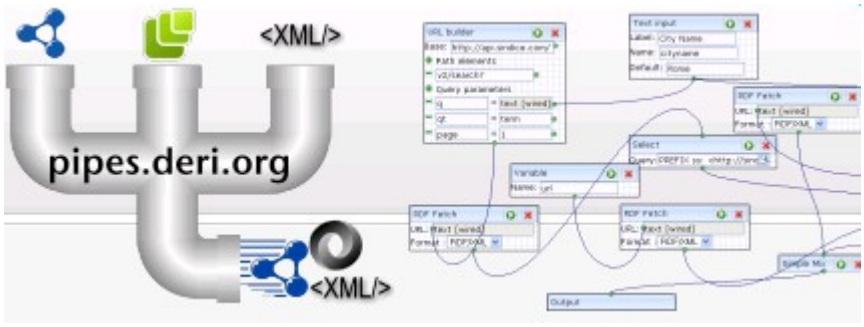
DERI Pipes



User Guide (v 0.6)

This brief guide intends to help developers install and begin using and developing with DERI Pipes. To find out more about the project see: http://pipes.deri.org

Please note that this is a functional, but early release of DERI Pipes. The release provides an opportunity for the community and industry to become familiar with Pipes and to contribute to the future direction of the project.

Table of Contents

1 Installation

This section describes how to download and install the DERI Pipes application

1.1 Pre-requisites

Before installing DERI Pipes you will need:

1. Java 1.6 or newer
2. A servlet container, such as Tomcat

1.2 Installation

1. Start tomcat
2. Download the current release of DERI Pipes (pipes-x.x.x.war) from sourceforge http://www.sourceforge.net/projects/swp
3. Rename the war file to pipes.war and copy it to the webapps folder. Tomcat will unpack the war file to create the pipes web application.

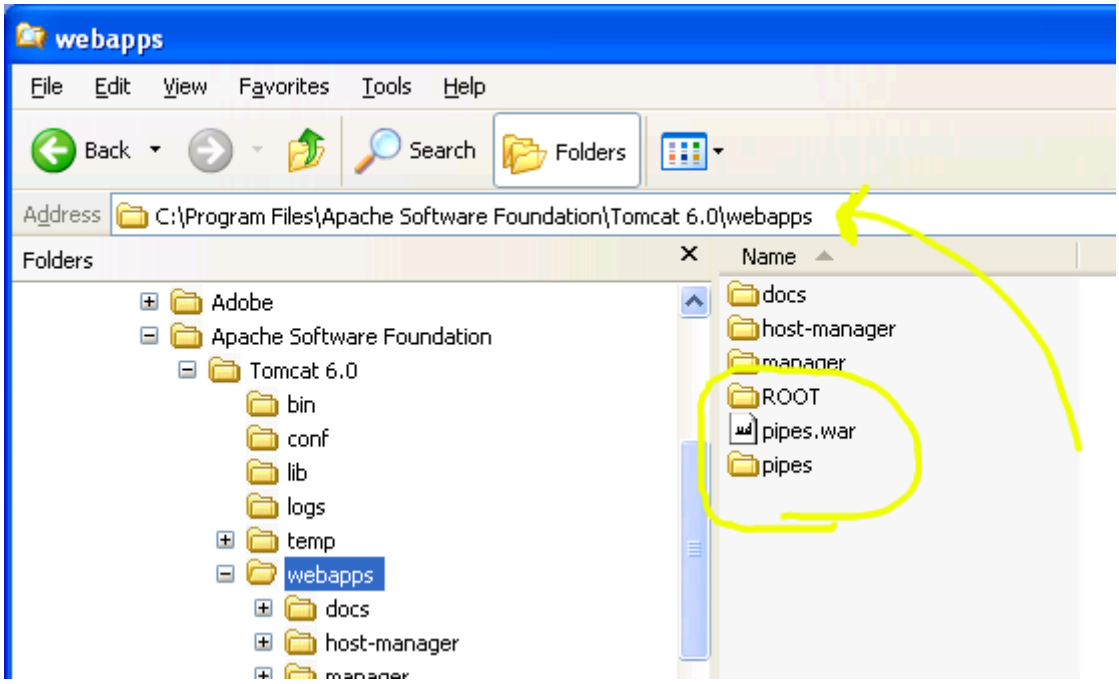


Illustration 1: Copying pipes.war to the Tomcat webapps folder

2 Using Pipes Web Application

2.1 Accessing Pipes

Depending on the configuration of your servlet engine, the pipes web application can be accessed at http://server:port/pipes/ . For a typical local installation of pipes in Tomcat, try http://localhost:8080/pipes/

2.2 Becoming Familiar With Pipes

Th following picture of the Pipes web interface may help the user become familiar with the primary operations of the Pipes user interface.

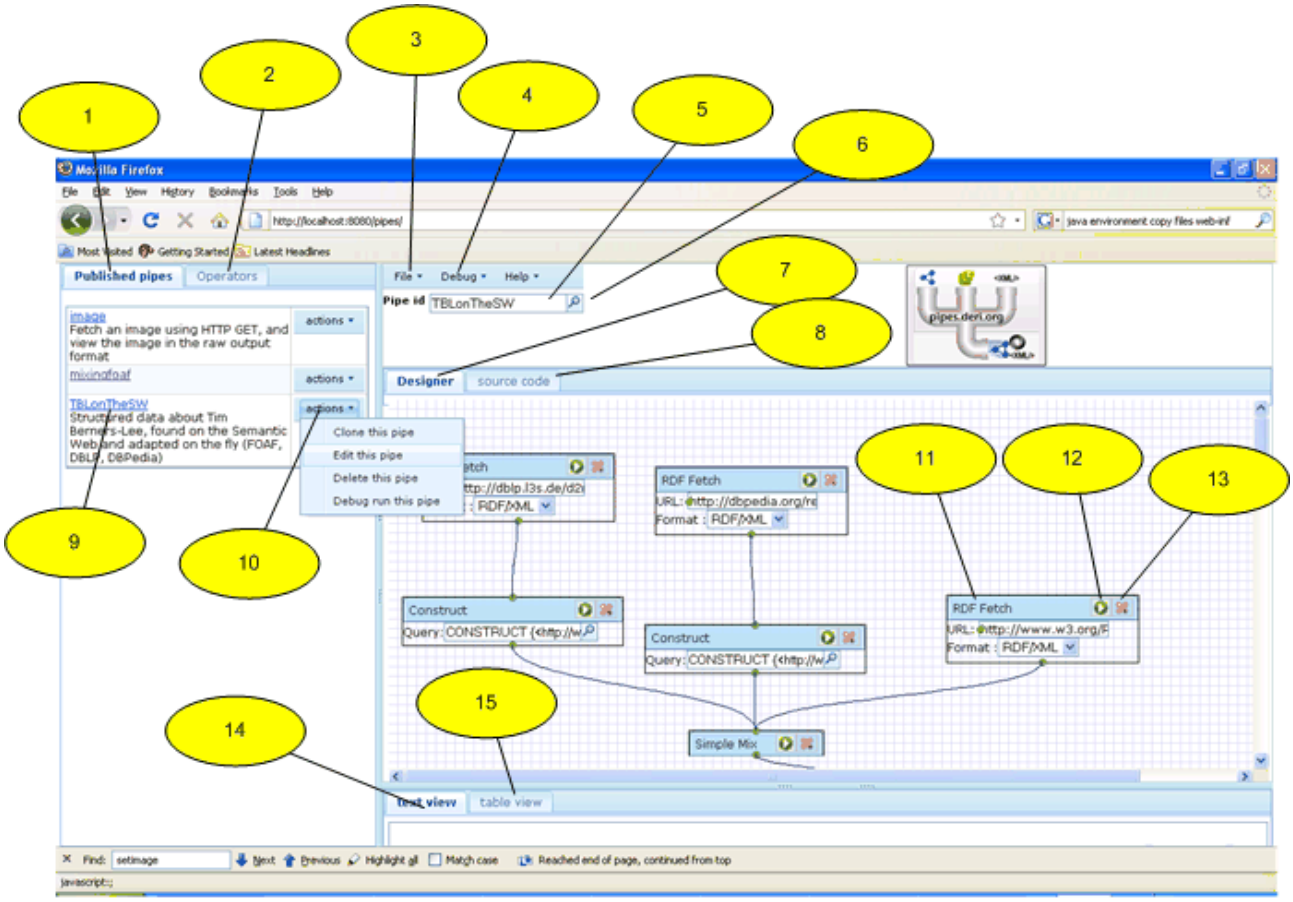


Illustration 2: DERI Pipes User Interface

A brief description of the marked items follows:

1. **Published Pipes tab**. Select this tab to run, edit or copy an existing pipe. A small number of example pipes are provided with the application.
2. **Operators tab**. Select this tab to view the available operators. Add a new operator onto your pipe application by dragging the chosen operator onto the design area.
3. **File menu item**. Use this menu to start a new pipe or to save your work.
4. **Debug menu item**. Use this menu to execute your pipe in debug mode. The output of the pipe execution is displayed at the lower part of the screen (see 14 and 15).
5. **Pipe ID text field**. This field displays the name of the pipe currently in the editor. Use this field to assign a name to a new pipe before saving.
6. **Pipe details glass**. Select this icon to view and edit the pipe ~~name~~ description and password.
7. **Designer tab**. Select this tab to modify the pipe in design mode.
8. **Source Code tab**. Select this tab to view the pipe xml source code to be executed.
9. **Saved Pipe execution link**. Click on the link to view the output of the pipe in html, RDF/XML or raw format.
10. **Actions context menu**. Choose an item from this menu to copy, edit, delete or debug a saved pipe.
11. **A Pipe Operator Node**. Enter the details required for the chosen operation.

12. **Pipe Operator Play Button**. Click this button to view the output of the pipe as far as this operator. The output is displayed in the debug window (see 14 and 15).
13. **Delete Operator button**. Click on this button to delete an operator.
14. **Debug Text View**. This tab displays the output of a debug operation in the raw form.
15. **Debug Table View**. This tab displays the output of the debug operation in tabular form.

2.3 Connecting Pipe Operators

Pipe Operators are connected by dragging the output port (at bottom center) of one operator to the input port (top center) of another. The pipe is complete when all required nodes have been added to the model, each connected to at least one other node. Each pipe model has one Output node which must be connected to in order for the model to be complete. The simplest pipe will always have at least two nodes.

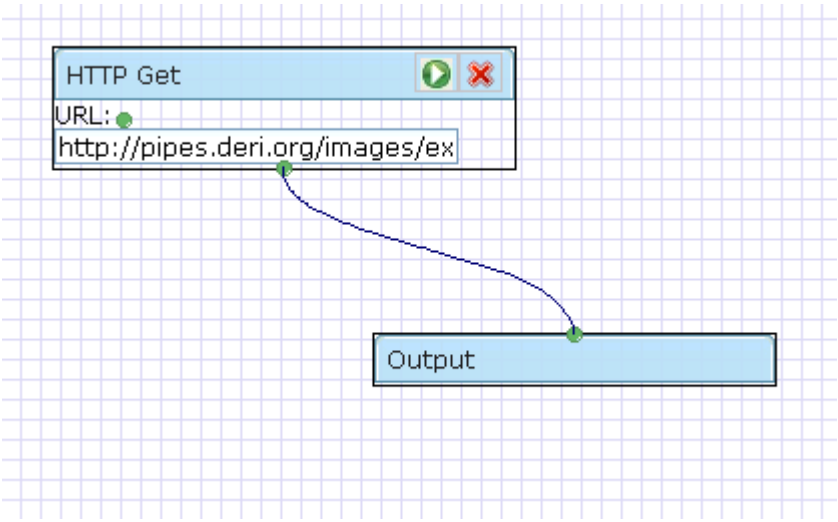


Illustration 3: Simple Pipe with two Nodes

Connections between Pipe Node Operators are made by dragging the output node of the operator to the input node of the next operator in the pipe. Note that not all pipe nodes are compatible. Some pipes expect input format of a particular type such as RDF, and will not accept other formats. If the connections between two pipes refuse to 'stick', this is likely because their formats are incompatible.

2.4 Viewing the Log Files

Pipes logs messages using slf4j logging framework. Look for pipes log messages in or around your other tomcat log files. Improved instructions on configuration of logging is intended for the next release.

3 Using Pipes on the command line

The DERI Pipes Engine allows pipes to be executed on the command line or embedded within an application.

To use pipes on the command-line, download and unpack the pipes-engine package.

Note: Pipes can be used to readline input from the STDIN input stream, by including a <stdin/> operator as a source in your pipe.

To invoke a pipe expecting name and place parameters, invoke as follows:

```
bin/pipes path/to/pipe/syntax/file name=Giovanni place=Galway
```

The remainder of this section shows by example how to execute a pipe on the command line. Here follows an example pipe reading from stdin. This requires 2 files, input.txt and pipe.txt which are shown below:

Command line is:

```
bin/pipes pipe.txt <input.txt
```

Output is:

```
<rdf:RDF xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
         xmlns:foaf='http://xmlns.com/foaf/0.1/'>
  <foaf:person rdf:about='http://example.com/Robert_Fuller'>
    <foaf:name>Robert Fuller</foaf:name>
    <foaf:firstName>Robert</foaf:firstName>
    <foaf:surname>Fuller</foaf:surname>
  </foaf:person>
  <foaf:person rdf:about='http://example.com/Giovanni_Tummarello'>
    <foaf:name>Giovanni Tummarello</foaf:name>
    <foaf:firstName>Giovanni</foaf:firstName>
    <foaf:surname>Tummarello</foaf:surname>
  </foaf:person>
  <foaf:person rdf:about='http://example.com/Dahn_Le_Pouch'>
    <foaf:name>Dahn Le Pouch</foaf:name>
    <foaf:firstName>Dahn</foaf:firstName>
    <foaf:surname>Le Pouch</foaf:surname>
  </foaf:person>
</rdf:RDF>
```

==========input.txt==========

```
Robert Fuller
Giovanni Tummarello
Dahn Le Pouch
```

==========

==========pipe.txt==========

```
<pipe>
  <code>
    <scripting>
      <language>groovy</language>
      <script>
import groovy.xml.MarkupBuilder
 def writer = new StringWriter()
 def xml = new MarkupBuilder(writer)
 xml.'rdf:RDF'('xmlns:rdf':'http://www.w3.org/1999/02/22-rdf-syntax-
ns#','xmlns:foaf':'http://xmlns.com/foaf/0.1/'){
```

```
    input.inputStream.eachLine{
     name -> parts=name.split(/\s/,2);
      xml.'foaf:person'('rdf:about':"http://example.com/${name.replaceAll('
','_')}"){
       'foaf:name'(name)
       'foaf:firstName'(parts[0])
       'foaf:surname'(parts[1])
      }
    }
  }
writer.toString();
       </script>
       <source>
         <stdin/>
       </source>
     </scripting>
   </code>
</pipe>
```

==========

4 Using Pipes Programatically

4.1 Executing Stored Pipes

Pipes provides a powerful and simple API to allow pipe development and execution from within the java applications. A good starting point for developers is to retrieve the pipes application from CVS on sourceforge, and to begin by studying some of the unit tests available.

In code a pipe can be created programatically and executed. Alternatively pipes can be retrieved from the pipe store for execution; this usefully allows pipes to be developed in the graphical environment but used other in non-web applications.

```
Engine engine = new Engine();
engine.setPipeStore(new FilePipeStore("test/data/pipe-library"));
PipeConfig config = engine.getPipeStore().getPipe("tblotw");
Pipe pipe = (Pipe) engine.parse(config.getSyntax());
ExecBuffer result = pipe.execute(engine.newContext());
ByteArrayOutputStream bout = new ByteArrayOutputStream();
result.stream(bout);
System.out.println("output: "+bout.toString("UTF-8"));
```

Text 1: Example of using Pipes Programatically to execute a saved pipe

Note that creating a Pipe Engine is a relatively expensive operation, however engines may be safely shared among threads of execution. Where practical an engine should be shared within an

application. A default engine is avaliable by calling Engine.getDefaultEngine();

4.2 Pipe XML Definitions

While a detailed explanation of the Pipe XML format is left for a later date, it is worth noting that there are two separate xml formats associated with the pipes. The first is a 'config' format which is used for rendering the pipe node operators in the GUI. The 'syntax' xml format is contains the definition of the pipes for the execution environment. While both formats are saved in the Pipe Store, only the syntax format is required in order to recreate a pipe for execution. In some cases where the graphical environment is not required, it may be desireable to save only the pipe syntax in order to minimise overheads involved in parsing the pipes. For efficiency pipes are parsed from XML into java objects using Xstream.

4.3 The Pipe Store

By default pipe XML definitions are stored one-per-file in a directory on the file system. In the case of many stored pipes these should be stored throughout a logical tree of sub-folders. A database implementation of the Pipe Store is also provided. Those wishing for an alternative implementation of the Pipe Store should impelment the org.deri.pipes.store.PipeStore interface. The PipeStore can be set by calling engine.setPipeStore(pipeStore)

5 Tutorial – Developing a new Pipe Operator

This section is intended to encourage developers to extend the functionality of DERI Pipes by creating new pipe Operators.

5.1 Overview

Conceptually there are two things which need to be done in order to create a new operator and make it available on the Pipes GUI. Firstly, and most simply, provide an implementation of the org.deri.pipes.core.Operator interface. Secondly, implement a class extending the abstract class org.deri.pipes.ui.PipeNode which represents the new Operator on the GUI and is responsible for loading to/from xml.

This section walks through the steps required to implement a simple HTTP GET Operator and add it to the GUI.

5.2 Implementing the Operator interface

The operator interface is very simple consisting of a single method:

```
public interface Operator{
          /**
           * Execute an operation and return the output.
           * @param context The current pipe execution context.
           * @return A buffer containing the results of the execution.
           * @throws Exception
           */
     public ExecBuffer execute(Context context) throws Exception;
}
```

Implementors should make note of the following guidelines:

1. Any variables not required as part of the pipe syntax should be declared transient in order to avoid serialization into xml.
2. To support multithreaded execution Operator should not maintain state information about the execution.
3. In order to maximise reusability, define Operators to specialise at a single operation.

Note regarding the ExecBuffer – There are currently 5 implementations of ExecBuffer which are:

1. SesameMemoryBuffer – Holds RDF into an in-memory model.
2. SesameTupleBuffer – Holds sqarql query results in in-memory.
3. StreamBuffer – Holds a reference to a URL which can be streamed OR holds string content in memory.
4. BinaryContentBuffer – Holds binary or text data in memory.
5. MultiExecBuffer – A collection of two or more ExecBuffers possibly executed in parallel.

It is important to choose the ExecBuffer implementation which fits most closely with the type of result produced by your operator. Our example HTTP GET Operator produces an BinaryContentBuffer because the format of the retrieved data is not known beforehand. Subsequent operators can be applied to the BinaryContentBuffer to transform the data from raw format into the desired output format.

Code for the HTTP GET Operator follows. A couple of items worthy of note are:

1. XstreamAlias annotation declares the xml element name for this class. (NB: Not strictly required since aliases are mapped in operatormapping.xml file)
2. The logger is defined as transient.
3. No state information is saved.

After implementing the class, an entry is created in the operatormapping.xml file:

```
<entry>
 <string>http-get</string>
 <java-class>org.deri.pipes.rdf.HttpGetBox</java-class>
</entry>
```

Text 3: Alias entry in operatormapping.xml

```
/**
 * Peri
 * Http
 * @aut
 *
 */
@XStrea
public
LoggerFactory.getLogger(HttpGetBox.class);
        String location;
        public String getLocation() {
                return location;
        }
        public void setLocation(String location) {
                this.location = location;
        }
        /* Perform a HTTP GET operation and return the
         * result in a BinaryContentBuffer.
         */
        @Override
        public ExecBuffer execute(Context context) throws Exception {
                HttpClient client = context.getHttpClient();
                GetMethod getMethod = new GetMethod(location);
                int response = client.executeMethod(getMethod);
                if(response != 200){
                        logger.warn("The http get request to
["+location+"] response code was  ["+response+"]");
                }

                BinaryContentBuffer buffer = new BinaryContentBuffer();
                buffer.setContent(getMethod.getResponseBody());

        buffer.setCharacterEncoding(getMethod.getRequestCharSet());
                Header contentType =
getMethod.getResponseHeader("Content-Type");
                if(contentType != null){
                        buffer.setContentType(contentType.getValue());
                }
                return buffer;
        }


}
```

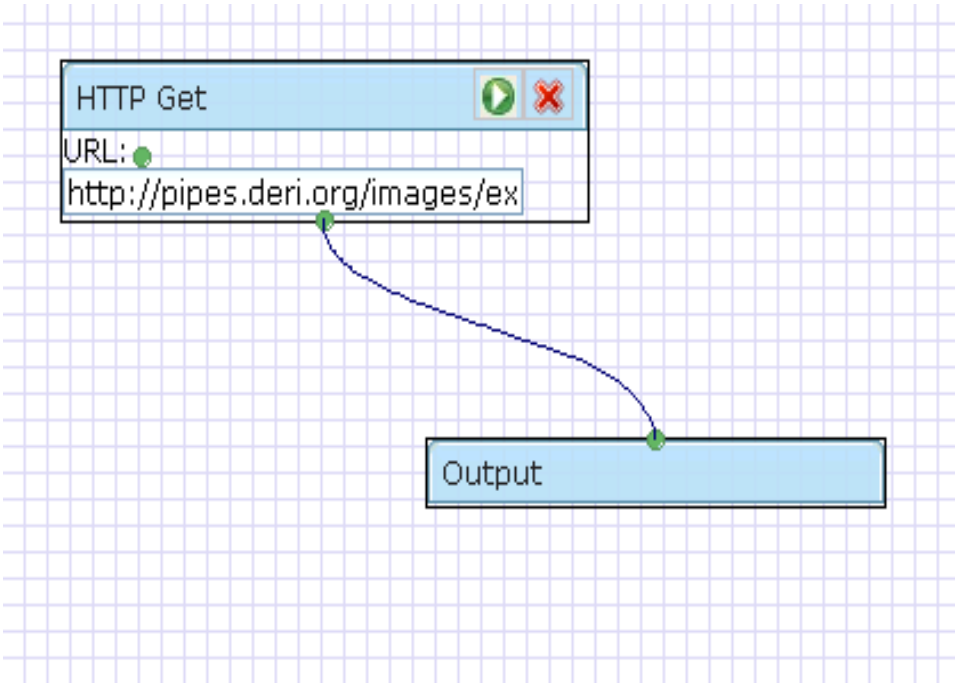
Text 2: Implementation of Operator Interface

Once implemented the serialization for the class can be verified in a unit test.

```
public void test() throws Exception{
   HttpGetBox get = new HttpGetBox();
   get.setLocation("http://www.deri.org/");
   String xml = engine.serialize(get);
   HttpGetBox get2 = (HttpGetBox)engine.parse(xml);
   assertEquals("wrong location", "http://www.deri.org", get2.getLocation());
}
```

5.3 Extending the PipeNode class

Unfortunatley this task remains a bit daunting and requires attention of the core developers in order to ease the burden required to provide new operators, nevertheless we demonstrate a simple gui component extending the PipeNode class for our HTTP GET operator. The finished GUI Element has the following appearance, here shown connected to Output. The node is quite simple, collecting only the URL location.



Full source code implementation of the class follows. Items of note include the ANYOUT port type which allows this node to be connected to classes expecting specific input types. Execution errors will occur when the connected operator's expectations are not met, for example if an Operator expecting rdf receives an image.

```
public class HttpGetNode extends InPipeNode implements
ConnectingInputNode{
          final transient Logger logger =
LoggerFactory.getLogger(FetchNode.class);
          protected Textbox urlTextbox=null;
          protected Port urlPort=null;

          public HttpGetNode(int x,int y){

          super(PipePortType.getPType(PipePortType.ANYOUT),x,y,230,60);
                    this.tagName="http-get";
                    wnd.setTitle("HTTP Get");
                    org.zkoss.zul.Label label=new org.zkoss.zul.Label("
URL: ");
         wnd.appendChild(label);
         urlTextbox =new Textbox();
         urlTextbox.setWidth("200px");
                    wnd.appendChild(urlTextbox);

          }


          protected void initialize(){
                    super.initialize();
                    urlPort =createPort(PipePortType.TEXTIN,35,36);
                    ((CustomPort)urlPort).setMaxFanIn(1);
          }

          public void onConnected(Port port){
                    urlTextbox.setValue("text [wired]");
                    urlTextbox.setReadonly(true);
          }

          public void onDisconnected(Port port){
                    urlTextbox.setValue("");
                    urlTextbox.setReadonly(false);
          }

          public void setURL(String url){
                    urlTextbox.setValue(url);
          }

          public Port getURLPort(){
                    return urlPort;
          }
```

The remaining two methods respectively serialise and parse the xml format for this operator:

```
		@Override
		public Node getSrcCode(Document doc,boolean config){
			if(getWorkspace()!=null){
				Element srcCode =
doc.createElement(tagName);
				if(config){
					super.setPosition(srcCode);
				}

				Element locElm
=doc.createElement("location");
				locElm.appendChild(getConnectedCode(doc,
urlTextbox, urlPort, config));
				srcCode.appendChild(locElm);
				return srcCode;
			}
			return null;
		}
		/**
		 * Creates a new HttpGetNode and adds it into the
configuration.
		 * @param elm The element defining this http-get
		 * @param wsp The PipeEditor workspace
		 * @return
		 */
		public static PipeNode loadConfig(Element elm,PipeEditor wsp)
{
			HttpGetNode node= new
HttpGetNode(Integer.parseInt(elm.getAttribute("x")),Integer.parseInt(e
lm.getAttribute("y")));
			wsp.addFigure(node);
			Element
locElm=XMLUtil.getFirstSubElementByName(elm, "location");
			node.loadConnectedConfig(locElm, node.urlPort,
node.urlTextbox);
			return node;
		}

}
```

Finally, once the HttpGetNode has been implemented, it needs to be referenced in the following two files:

1. PipeNodeFactory.groovy (2 places) to produce the correct PipeNode when the http-get element is encountered:

```
case "htmlfetch":
    return new  HTMLFetchNode(x,y);
case "http-get":
    return new HttpGetNode(x,y);
case "parameter":

...
case "htmlfetch":
   return HTMLFetchNode.loadConfig(element,pipeEditor);
 case "http-get":
   return HttpGetNode.loadConfig(element,pipeEditor);
 case "parameter":
```

2. index.zul one place to have the HTTP GET name appear on the Operators tab.

```
<treeitem>
    <treerow>
      <treecell id="http-get"  draggable="true"
             label="HTTP GET" />
      </treerow>
</treeitem>
```

Once these changes have been made the new PipeNode is available in the Pipes web interface.

6 Conclusion

The DERI Pipes application has matured and is ready for deployment. The web application is simple to install and runs 'out of the box'. A programming API allows developers to embed the DERI Pipes Engine directly into custom applications.

The application can be easily extended by adding new Operators. The Operators are made available in the Pipes GUI by extending the PipeNode class and modifying the configuration slightly.

7 Frequently Asked Questions

Q. Are DERI Pipes limited to RDF based operations?

A. No, Operators can now be defined to operate upon and return any content types. Plans for operators to perform operations on text and images are planned and contributions are welcome.

Q. What are the licensing terms for DERI Pipes.

A. This remains to be determined. A BSD Style license is preferred, however the situation is complicated by some dependency on GPL Licensed components. Corporate lawyers from around the world are studying the situation carefully.

Q. I have made a very useful pipe, can you add it to the core project?

A. Submissions are most welcome and will be reviewed on a case by case basis.

Q. I found a bug, what should I do?

A. Please check Jira and the mailing lists to see if this is an old issue. If not, please register a new issue in Jira at: http://dev.deri.org/

Q. Is there a command line interface for running stored pipes from the shell?

A. This will be provided in an upcoming release.

Q. Is there a way to use a java object held in memory as a Pipes source in such a way that the pipe operation can be used with the in-memory object as input?

A. This will be provided in the very near future.

Q. Can Pipes be configured using Spring Framework?
A. This is not done currently but will be provided in a future release.

Q. Can operators be implemented in languages other than Java?

A. To do this a java wrapper of the Operator class is required. The wrapper can delegate to the underlying implementation in an appropriate manner. Note also that a Scripting implementation is provided which allows execution of scripts in Groovy, Python etc. This operator is not available in the GUI for security reasons.

Q. Is there more documentation available?

A. Some further documentation can be found on the pipes website: http://pipes.deri.org

8 References

Tomcat	http://tomcat.apache.org/
Java 1.6	http://java.sun.com/javase/downloads/index.jsp
ZK Diagram	http://www.integratedmodelling.org/software/zk/zkdiagram.html
Slf4j	http://www.slf4j.org/
Spring	http://www.springsource.org/
DERI Pipes Website	http://pipes.deri.org
Pipes Sourceforge Page	http://sourceforge.net/projects/semanticwebpipe/
Pipes JIRA	https://dev.deri.ie/jira/browse/SWP

Robert Fuller,

robert.fuller@deri.org

Feb 10th 2009