8 October 2003

Topologi Application Note

Running OmniMark and Perl Scripts from Topologi

Rick Jelliffe

Guide for Developers: Draft only

This application note give details on how to add menu items (plugin scripts) to the Topologi Collaborative Markup Editor to run text processing scripts, for example to invoke Perl or OmniMark.

Topologi plugins use the BeanShell language, which is an interpreted version of Java with access to the full Java API. All Topologi plugins are functions which accept, process or use a *PluginData* object. There is no direct access to the current documents being edited nor the GUI. However, the *PluginData* object makes available useful parts of the Topologi application, in particular the Options panel, the File Chooser, and the Status Bar.

See the document *Topologi Customization Manual* for full API details.

Various kinds of scripts can be provided for the different kinds of menus. This Application Note only deals with *processing* scripts, which take the document's text (or the selected text in the current document) and then processes it, and then replaces the document (or selection) with that text. If you want to transform the document and open it as a new document, use the *preview* scripting.

Stub

The following code fragment gives the stub which any processing plugin needs.

```
import java.io.*;
import com.topologi.tme1.io.plugins.*;
import com.topologi.tme1.services.statusbar.*;
import com.topologi.tme1.io.worklistmodel.*;

executePlugIn(PlugInData p, PlugInResultHandler handler) {

   if (p.getData() == null || p.getData().toString().equals("")) {
      p.setError("No data was sent to the script");
      handler.handleScriptResult(p);
      return;
   }
   String inString = p.getData().toString();
   StringBuffer outBuff = new StringBuffer();

  // PROCESSING GOES HERE


    if (outBuff == null) {
         // user cancelled
         p.setStatus(p.CANCELLED);
         handler.handleScriptResult(p);
         return;
    }
    p.setData(outBuff);
    handler.handleScriptResult(p);
}

boolean isEnabled(TopologiDocument td) {
   // FUNCTION GOES HERE
  return true;
}

return (PlugInScript) this;
```

IMPORT

You should put imports statements just as you would for any Java program.

EXECUTEPLUGIN()

This is the method that is called when the menu item is selected. The first test should be for when there has been no data passed back in the *PlugInData* object.

At the end, the script sets the new value into the *p* object, then returns that to the appropriate plugin handler.

Note that the plugin data sends and recieves *StringBuffer*s rather than *String*s. This can reduce (or at least delay) the amount of copying required.

ISENABLED()

The *isEnabled()* method sets whether the menu item is enabled. Everytime the current document is changed to a different file, or the notation is changed, this method is changed.

The *TopologiDocument* that is passed as the argument contains much information about the document. Persistent data can be added to the *TopologiDocument*.

Here is an example of an *isEnabled()* method, which enables the menu for XML and SGML only:

```
boolean isEnabled(TopologiDocument td) {
   // Notation must be XML
   if (!(td.getNotation().equals("xml") || td.getNotation().equals("sgm")
        || td.getNotation().equals("sgml") || td.getNotation().equals("ecs")  )) {
      return false;
   }
   return true;
}
```

RETURN

The final return statement is run when the plugin is initially loaded. If you plugin needs to do any preinitialization (e.g. and keep the results in the Options object) this can be done here.

Running the Command

There are two versions of the method to process commands, *runCommand()*.

One version uses the same temporary file for reading and writing: it is suitable for commands which read and write to the same file. The other version is suitable for commands which read and write to different temporary files.

```
try {
    FastUUID f = new FastUUID();
    File tmpFile = File.createTempFile(f.toString(), ".txt");
    tmpFile.deleteOnExit();
    com.topologi.lib.RunCommand.runCommand(
            command, inString, outBuff, tmpFile, readEncoding, writeEncoding,
            p.getStatusLine() );

} catch (IOException x) {
    // couldn't exec command
    p.getStatusLine().handleMessage(
            new StatusMessage(
                    “Could not create the file " + tmpFile.toString(),
                    StatusMessage.TIP,
                    StatusMessage.GENERAL_SERVICE,
                    StatusMessage.GENERAL_PRIORITY,
                    (short) 500));
}
```

This is the two-file version:

```
try {
    FastUUID f = new FastUUID();
    File tmpFile = File.createTempFile(f.toString(), ".txt");
    tmpFile.deleteOnExit();

    File tmpFile2 = File.createTempFile(f.toString(), ".txt");
    tmpFile2.deleteOnExit();


    com.topologi.lib.RunCommand.runCommand(
            command, inString, outBuff, tmpFile, tmpFile2, readEncoding, writeEncoding,
            p.getStatusLine() );

} catch (IOException x) {
    // couldn't exec command
    p.getStatusLine().handleMessage(
            new StatusMessage(
                    “Could not create the file " + tmpFile.toString(),
```

```
                    StatusMessage.TIP,
                    StatusMessage.GENERAL_SERVICE,
                    StatusMessage.GENERAL_PRIORITY,
                    (short) 500));
}
```

Adding a Dialog Box

Sometimes you may need to query the user. Here is an example of a dialog box:

```
String noString = "No";
String yesString = "Yes";

JOptionPane option1 = new JOptionPane("The label on the buttons");
option1.setOptions(new String[] { noString, yesString });
option1.setInitialValue(yesString);

JDialog dialog1 = option2.createDialog(
      p.getServices().getReferenceFrame(Main.APP_MARKUPEDITOR),
     "The label on the window" );

dialog2.show();

if ((option2.getValue() != null) && ((String) option2.getValue()).equals(yesString)) {
  // YES CODE HERE
} else {
  // NO OR CANCEL CODE HERE
}
```

To use a dialog box, you also need to add *import javax.swing.*;*

Sending a Message to the Status Line

The status bar gives messages concerning the progress, success or failure of a script. It can be a TIP, WARNING or ERROR. The number at the end is the default time to display it in ms. Typically this is a fairly short time.

```
p.getStatusLine().handleMessage(
   new StatusMessage(
     "Some Message",
     StatusMessage.TIP,
     StatusMessage.GENERAL_SERVICE,
     StatusMessage.GENERAL_PRIORITY,
     (short) 500));
```

Shortcut for Open Plugins

The editor provides a shortcut for processing files when using the Open plugins, which cascade.

First you set the following properties on the document data:

- *run-script-on-transcoded-data*
- *run-script-on-transcoded-data-write-encoding*
- *run-script-on-transcoded-data-read-encoding*

These are strings. For example,

```
p.getDocumentData().setProperty("run-script-on-transcoded-data", command);
```

Then, as the last statement in *executePlugin()* you pass control to the text import transcoder, which in turn runs this command.

```
TranscodingRunner runner =
        new TranscodingRunner(
              p.getServices().getReferenceFrame(Main.APP_MARKUPEDITOR),
              handler);
runner.transcodingCall(p, encoding, false, 0x1000, 100);
```