Process Management with SmartFrog

16 Aug 2007

1 Introduction

SmartFrog is a framework for writing distributed configuration systems, configuring nodes or other devices as determined by a description of the desired state of the system. As such, whether the system being configured is a Windows platform, a Linux platform or some other operating environment, much of the low level tasks to be carried out will involve executing commands or running daemons of the underlying operating system to achieve the desired configuration. This report shows how the SmartFrog system can make this process easy through the use of the process management and shell scripting components provided as part of the core release.

Note that this document assumes a working knowledge of SmartFrog; if basic knowledge of this is required, the tutorial and reference manuals should provide the kind of information required.

It is always possible for a component programmer to resort to the use of the Java command execution libraries, however these are not easy to use. Better is to provide a series of components that are heavily tested and hide much of the unpleasantness of dealing with processes and shells, making for a quicker and more reliable way to handle many configuration tasks.

The document does not deal with how to write well-formed components that interact directly through the Java libraries to create processes or shells. This is not easy, due to the complexity of the model provided in these libraries.

The document is structured into three main sections:

1. process management: how to run a process using the process management component,
2. shells: how to run a a specific shell (a special case of a process) and interact with it in a form more suited to shells,
3. scripts: how to represent interactions with a shell as a component.

In each case, as appropriate, the discussion of each aspect is split into two parts: the component model and the programming model. The first, the component model, shows how to describe a component so that it may be deployed and configured as required. The second, the programming model, examines how a component can be written that interacts through interfaces exported by the components. Scripts do not export an interface, so this is not appropriate for this component.

Before this discussion, there is a section covering some of the general aspects of writing components and those parts that are common between these three component types.

2 Common Component Aspects

This section covers some of the general points around the description of the components and their semantics.

2.1 Logging

All the components log their activity to a log interface. They use the logging `sfLog()` method with the default logger. A specific named logger can be used by defining the `sfLog` attribute to the name of that logger.

The log messages themselves contain an identifier for the component that is issuing each log message. By default this is the `sfCompleteName()` of the component but if a different name is required, this can be set by providing the optional “name” attribute.

2.2 Commands

An important concept to grasp is the uniform way in which these components represent processes to manage – this is used directly by the user when configuring the process management components, but is also used to describe the shell process (but not the commands sent to the shell) in the shell component. In the latter case, normally users will not see this as descriptions have been provided for many of the common shells – such as Bash under Linux and the Windows CMD shell under XP – which have these predefined.

A command may have a number of attributes:

- the working directory
- a set of environmental variables
- the command to be executed

These are described according to the following schema, and basic wrapping component. These are defined in a file contained in sfServices.jar:, namely "`org/smartfrog/services/shellscript/components.sf`"

```
// schema and core component for a command
ExecSchema extends Schema {
    cmd extends Vector;
    dir extends OptionalString;
    envProperties extends OptionalVector;
}

Exec extends LAZY {
  execSchema extends ExecSchema;
}
```

The working directory (`dir`) is a string representing the directory name, the environment is a vector of strings, each representing a variable binding (see example below) and the command is a vector of strings, each representing one token of the command-line (again see example).

Given this, an example of a valid command might be:

```
// example of the use of a command
#include ".../shellscript/components.sf"

myProcess extends Process {
   exec extends LAZY Exec {
      workingDirectory ".";
      envProperties ["sfHome=/usr/sf", "tempDir=/tmp"];
      cmd ["sfStart", "localhost", "foo", "foobar.sf"];
   }
}
```

Now a description of a command is always data, not a component definition, so it will always occur as an `extends LAZY` or `extends DATA` declaration. Furthermore, in both the process management component and the shell component, the attribute which holds the command definition is called "exec" as in the example above.

2.3 Tracking execution status

The process manager and the shell, the two components that manage processes as determined by the command descriptions, both track the status of their process. They both provide attributes as part of the component context which describe the current state:

- `execExitCode` – the exit code of the command the last time the process exited (died).
- `numberOfExecs` – the number of times that the process has been started
- `execExitCodes` – a vector of integers, in the 0th position is the number of execs, and in all others (if they exist) the exit codes of all the process exit codes in order.

In actual fact, these attributes are only useful if the description of the process management indicates that the process should be restarted (or at least not terminated) on process failure, and this never happens in the case of a shell. However, initial values of these attributes do appear it the run-time context of the shell component, hence the need to document their existence.

3 Process management

The process management component provides a simple management model for processes, with the ability to access the various I/O streams of the process through a local Java interface. In its most basic form, the control of the process is a simple definition of the command to execute and a set of attributes that control how the life

cycle of the component mirrors (or otherwise) by the life cycle of the process it is managing.

3.1 The component model

The `ProcessManager` component has the following schema and basic default component.

```
// schema and wrapping component
ProcessSchema extends Schema {
  detach extends OptionalBoolean;
  restart extends OptionalBoolean;
  terminate extends OptionalBoolean;
  autoStart extends OptionalBoolean;
  string extends OptionalString;
}


Process extends Prim {
   execShema extends ProcessSchema;
   sfClass "...shellscript.SFProcessExecutionImpl";
   //retart false;
   //terminate false;
   //detach false;
   //auotStart true;

   exec extends LAZY Exec;
   sfExport false;
}
```

The component starts the process by executing the command – it does this during the `sfDeploy` phase of the life cycle. If, at any time, the process should terminate, the following actions will be taken.

if restart is true, the process will be restarted if it ever fails unexpectedly, if it is false it will be left dead until told (programmatically) to restart. (Default false.) Note that the component does not deal properly with repeat failures within a short time interval – in other words, currently the component will simply continue to try to restart the process every time a failure occurs. There is no limit which would cause an alternative action – such as component termination.

1. If `terminate` is true, the component managing the process will carry out an abnormal termination if the process fails – this dominates over any restart that may be required. If it is false, it will not terminate but remain in existence, restarting the process if this is appropriate (Default: true.)
2. if `detach` is true, the component detaches itself from the parent before any abnormal termination. If false, any termination will propagate to the parent component if one exists. (Default false.)
3. if `autoStart` is set to false, the process will not start automatically at the time of deploying the component, but it will need to be started using the programmatic interface. If true, it will start automatically (Default true).

Note that unless the programmatic interface is in use, the component by default logs all the output from the process to the components standard log. This can be set as normal by using the `sfLog` attribute.

Note that the shell component must not be exported to RMI as it has a non-remote interface that has to be accessed through linking. This means that there are some restrictions in the use of a shell. In particular, if it is to be deployed remotely, it must be wrapped in a remoteable component such as a `Compound`. Secondly, some of the remote management components may not work correctly – typically generating a serialization exception when they try to obtain a remote reference.

3.2 The programming model

The component offers a simple interface to control the process managed by the process. This interface is defined as follows.

```
package org.smartfrog.services.shellscript;

import java.io.InputStream;
import java.io.OutputStream;
```

```
public interface SFProcessExecution extends SFExecution {
   public InputStream getStdOutStream();
   public InputStream getStdErrStream();
   public OutputStream getStdInpStream();
   public void kill();
   public void restart();
   public boolean isRunning();
}
```

This methods have the following effect on the underlying semantics of the component. Starting with the two process control methods:

- The process `kill()` method terminates the process in a controlled way – this is not considered to be a failure and so the life cycle attributes are not used at this point.
- The process `restart()` method will kill the process is necessary, then (re)run the command as in the initial deploy. Failure of this will be treated as any other failure in the process.
- The `isRunning()` method returns a boolean describing whether the subprocess is running or not (terminated, failed or not started).

The get stream methods enable another component to interact with the process to signal it in some way, or to monitor its output to identify some events. Using them can be dangerous and difficult, and in particular if the `stdOut` and `stdErr` streams are accessed, the user must make sure that they are continuously emptied of data otherwise JVM lock-up can occur. If the streams are never obtained, the component creates threads that empty the streams and log the output found, once the streams have been obtained, control of them can never be regenerated.

Note that the failure (or other termination) of a process will result in the streams closing and the waiting threads being notified with an `InterruptedIOException`. At this point if the process is restarted, either by the process manager or by another component via the restart method described above, the new streams must be obtained and handled properly as described above. The component does not create new handlers for these streams.

Other data, such as the exit codes of the process if it fails, can be obtained by resolving references to the attributes `execExitCode` et al. As described above.

4 Shells

A specific type of process which is particularly useful for many configuration tasks is the shell component and the related component that has been created to submit scripts to the shell.

The shell component differs from a normal process component in that rather than accessing the streams directly, the script component (or any other component wishing to send scripts to the shell) use a command-based interface – submitting a set of command lines, and receiving back the associated lines of output. This is done using an asynchronous model of submitting lines to be executed and later being able to collect or wait for the results.

Unlike the generic process component, the shell component fixes some of the options. For example, it always starts the shell on component deployment and it always terminates if the shell exits. The rationale for this is that a shell carries state from previous scripts – for example the definition of environmental variables. Consequently, restarting a shell can lead to errors. If restarting is required, the shell can be wrapped in a restarting work flow component. Also, the component never detaches itself on termination, but will always propagate its failure.

4.1 The component model

The shell component defines the shell command that runs the shell process it is managing. In addition, it has a small number of additional attributes, mostly to do with differences between different shells running on different platforms.

The model for the component is that it can be given a number of scripts to run by either using the programmatic interface or using the Script component described below. In order to identify the boundary of the output of the

various scripts, the component adds an “echo” command between them. The output of this echo is determined by the component and it uses this to recognize these boundaries of the shell output. In addition, in order to extract the termination code of the last command, the component needs to be given the definition of the environment variable that does so - this is used in the delimiting echo to extract this.

Users need not be aware of the most common instances of these, as some predefined shell component descriptions are provided with the core definition. However, should a new shell be encountered, this may be set in the configuration description of a shell component.

The schema for this divided up as follows:

1. define the exec structure for the desired shell (examples here of Bash and the WinXP cmd shell.
2. define a shell as a prim parameterized by the appropriate exec for the shell.

```
ShellExecSchema extends ExecSchema {
    lineSeparator extends OptionalString;

    echoCmd extends OptionalString;
    echoErrorCmd extends OptionalString;
}

ShellExec extends Exec {
  execShema extends ShellExecSchema;
  echoCmd "echo";
}

BashExec extends ShellExec{
  cmd  ["/etc/bash"];
  echoErrorCmd "$?";
}

WinXPExec extends ShellExec {
   cmd    ["cmd.exe"];
   echoErrorCmd "%ERRORLEVEL%";
}
```

Note that a shell exec structure is extended by a number of additional attributes, in particular the concept of an line separator (by default obtained from the Java properties relating to the local platform, but it can be set explicitly), an `echoCmd` which defines a shell command that echoes to `stdout` a given string, and a `echoErrorCmd` string which, when echoed using the shell, returns the exit code of the last command executed.

Given these, it is possible to define the second part, the specific shells.

```
ShellSchema extends Schema {
    name extends OptionalString;
    ID extends Long;
    exec extends CD;
}

Shell extends Prim {
   shellSchema extends ShellSchema;
   sfClass "...shellscript.SFScriptExecutionImpl";
   exec extends LAZY Exec;
   sfExport false;
}

BashShell extends Shell {
   exec extends LAZY BashExec;
}

WinXPShell extends Shell {
   exec extends LAZY WinXPExec;
}
```

Note that the shell component must not be exported to RMI as it has a non-remote interface that has to be accessed through linking. This means that there are some restrictions in the use of a shell. In particular, if it is to be deployed remotely, it must be wrapped in a remoteable component such as a `Compound`. Secondly, some of the remote management components may not work correctly – typically generating a serialization exception when they try to obtain a remote reference.

This schema defines all the fields required for specialization of the component in various ways:

- the `cmd` attribute is used to define the program that will be run as part of the component (in most cases this will be a shell).
- The `dir` attribute defines the working directory for the command
- the `envProperties` defines environmental properties that should exist in the environment in which the shell should execute
- the line separator defines the characters to be used to separate the data sent to the program through the standard input (in most cases the commands sent to the shell). If not provided, this is obtained from the Java system property `line.separator` set on each platform.

4.2 The programming model

This is the most complex part of the use of the built-in shell script support within SmartFrog. If appropriate it is in general easier to use the Script component described below, though it is not always sufficient.

The shell component provides a local interface "..`shellscript.ScriptExecution`". This interface provides a line-oriented command interface to the shell component. Many components may use the shell to interleave their commands, with a locking mechanism provided to ensure that the sets of commands that must be implemented without interleaving can be guaranteed.

The model is asynchronous, in that a set of commands can be submitted to the shell, then at a later time the submitter may wait for the results or see if the results are ready .

The basic interface is defined as follows:

```
public interface ScriptExecution {
   public ScriptLock lockShell(long timeout)
        throws SmartFrogException;

   public ScriptResults execute(List commands,
                                ScriptLock lock)
        throws SmartFrogException;

   public ScriptResults execute(String command,
                                ScriptLock lock)
        throws SmartFrogException;

   public void releaseShell(ScriptLock lock)
        throws SmartFrogException;



   public ScriptResults execute(List commands,
                                long timeout)
        throws SmartFrogException;

   public ScriptResults execute(String command,
                                long timeout)
        throws SmartFrogException;
}
```

The interface can be divided into two groups – the first four methods use the shell locking mechanism, whereas the last two are single atomic actions on the shell. In both cases, the interface supports either supplying a single command or a list of commands, each of which will be separated in the shell by the `lineSeparator` character.

Note that in all cases, submitting a command (using one of the execute methods) consists of adding the command to the `stdIn` queue of the shell process, and that the fact that the method that submits a command returns in no way implies that the shell has started to execute the command. Indeed in general it may be queued behind a number of other commands from previous invocations of the execute methods. Furthermore, the method may block until the stream has room for the command to be added. This is not included in the timeouts associated with obtaining a lock.

The model for use of the locking methods is to request a lock, submit one or more execute commands (either

single or list), then release the lock. In the execute methods and the release lock method, the correct lock-object must be provided to ensure that method is being associated with the right lock operation.

An example code snippet is as follows:

```
import ...shellscript.ScriptExecution;
import ...shellscript.ScriptResults;
import ...shellscript.ScriptLock;

/*
* get an instance of a script component (e.g. via a LAZY
* link; in this example the attribute “shell”
*/
ScriptExecution se =
             (ScriptExection) sfResolveHere(“shell”);


/*
* obtain the lock, run a set of commands,
* release the lock
*/
try {
    ScriptLock sl = es.lockShell(-1);
    ScriptResults res1 = se.execute(“aVar=*.jar”, sl);
    ScriptResults res2 = se.execute(“aCommand $aVar”, sl);
} finally {
    if (sl != null) se.releaseShell(sl);
}

/* run a single command */
ScriptResults res3 = es.execute(“someCommand”, -1);
```

Note that in the `lockShell` method and the single command execute method there is a `timeout`. This `timeout` is simply to obtain the lock (implicit in the single method execute), not to start executing the command. The values for the `timeout` are

- 0: immediate return or throw an exception,
- -1: wait forever if necessary,
- anything else is waiting for a maximum of the specified number of milliseconds. The example above will wait forever for the lock to be released.

As the lock is not automatically released if an exception is generated, the `releaseLock` should be called within a `finally {}` clause to ensure that it is released.

Given that the command is submitted to the shell (though possibly queued behind others) when the execute method is complete, the next issue is how to collect the output of the commands.

Internally, the shell component adds an echo command between each execute method command or list of commands. In this way it can recognize the boundary of the output data that is generated by each individual execute method call.

This data is collected into two lists of strings, one string per line of output, one list for each of the two output streams. Also, the command boundary echo obtains the exit code for the last of the commands executed as part of that execute method. Both of these lists and that exit code can be obtained from the `ScriptResult` object.

However, the object does not necessarily have the results ready immediately – they may finally be available at some point in the future after the commands reach the front of the shell's buffer and execute. Consequently, the programming model provides two methods: waiting for the results to be available, with a timeout, or inspecting whether they are ready yet.

This is defined in the interface `ScriptResults`:

```
public interface ScriptResults {
   public boolean resultsReady();

   public ComponentDescription waitForResults(long timeout)
        throws SmartFrogException;
```

```
}
```

As with the execute methods, the timeouts are 0: immediate return, -1: wait forever, other values in milliseconds.

The component description returned contains the data from the result, in appropriately named attributes, These attributes are:

- code – the result code of the last line of the commands
- `stdout` – the list of lines output onto `stdout` by this set of commands
- `stderr` – the list of lines output onto `stderr` by this set of commands

Note that due to the way in which the output from the various commands are delineated (using an echo), and command which results in a background task may have its output appended to the wrong ScriptResult(s).

An example of a code snippet to make use of the results, following on from the previous example, is as follows:

```
// wait forever for the script to execute and complete
ComponentDescription res = res1.waitForResults(-1);

if (res.sfResolve("code", 0, true) == 0) {

    // the script completed properly (code 0)
    if (((List)res.sfResolve("stderr")).size() == 0) {

        // there was no error output
        List stdout = (List)res.sfResolve("stderr");
        for (Iterator lines = stdout.iterator(); ....) {
             // print  each output line
             System.out.println(lines.next());
        }

    } else {
        // there was some error output
        throw ...
    }

} else {
    // the script failed
    throw ...
}
```

Clearly the result processing could be quite complex, with the likely need to pattern-match over possible error messages, non-zero codes indicating possible errors, and so on. However this complexity is purely dependent on the complexity of the application's scripts and there is little that the supporting components can do to ease this. It is easier, however, than hooking directly into the streams of a process or having to do the management of the process oneself.

5 Scripts

A script is an easy component-oriented way of using the shell component which avoids the use of the programming interface. Indeed the scripts to execute are simply strings, or lists of strings, given as attributes to the component. Furthermore, the component can be used as a traditional component with the ability to run scripts during each of its life cycle phases, or it can be used as part of a work flow – running a script and then terminating.

A script component does not offer any run-time interface to other components, it is merely a component wrapper for the programming model provided by the shell to which it it linked.

5.1 The component model

In addition to the definition of the `scriptExecution` attribute which is a `LAZY` reference to the shell component, the component defines three script attributes: `deployScript`, `startScript` and `terminateScript`. Each of these is a vector of commands to send to the shell during the associated lifecycle phase. In addition, a boolean attribute `autoTerminate` can be set to true if the component should automatically self-terminate after execution

has completed – this is useful in a workflow.

The definitions in the components.sf file is as follows:

```
/** schema for the script component */
// first the schema for a single script entry
OptionalScript extends Optional {
    class ["java.lang.String",
           "java.util.Vector",
           "...componentdescription.ComponentDescription"];
}

// now the full schema with its set of lifecycle scripts
ScriptSchema extends Schema {
    shell extends Reference;

    deployScript extends OptionalScript;
    startScript extends OptionalScript;
    terminateScript extends OptionalScript;

    autoTerminate extends Boolean;
}

Script extends Prim {

    schema extends ScriptSchema;
    autoTerminate false;
}
```

Given these definitions, the semantics are:

- During the relevant lifecycle, the commands defined (if any) in the appropriate attribute are executed by the shell to completion.
- If, whist executing the deploy or start scripts the exit code for the last command is 0, the vector of commands is assumed to have executed correctly. If it is other than 0, a termination is triggered to indicate that an error occurred in the execution of the script.

  This will terminate the component in the way that a lifecycle method failing would do in any other component. The resultant abnormal termination record will contain details of the script execution. The error is also logged.
- If an error occurs during the termination script, no action is taken (apart from logging the error).
- If the `autoTerminate` attribute is true, after executing the start script the component will terminate with a normal termination record.

Now the definition of a script needs some explanation. A script can come in three forms:

1. a string, which is the command to send to the shell
2. a vector of strings, sent in one locked interaction, with each string being a single command.
3. a component description, with each attribute being a string, or a vector of strings or indeed a component description. The description is traversed top-down, depth-first. The whole is considered as a single locked

An example of the use of the Script component in use in conjunction with the required shell script component is shown below

```
#include "org/smartfrog/services/shellscript/components.sf";

sfConfig extends Compound {
    aShell extends BashShell;
    dns extends Script {
        shell LAZY aShell;
        deployScript ["/usr/bin/named ..."];
        terminateScript ["killall named"];
    }
    dhcp extends Script {
        shell LAZY aShell;
        deployScript ["..."];
```

```
        terminateScript ["..."];
    }
}
```

Note that the two script components assume that they are deployed after the shell component to ensure that the script is already executing by the time the scripts are run. Also, the shell and the scripts must be on the same host since the interface offered by the shell is local only.