Groovy support in SmartFrog

2 Feb 2011

1 Introduction

This document covers Javax scripting and Groovy support in SmartFrog

BeanShell support has long been a feature in SmartFrog. The javax.scripting components are the successor to this, taking advantage of the fact that JavaScript is built in to Java6 -now scripting is available without any extra dependencies.

The scripting components in the Groovy package represent an update of the original components, with some extra features. They are where future development will take place -feel free to file feature requests

2 Using

Add the sf-groovy libraries (and dependencies) to the classpath of the SmartFrog Daemon and command line tools.

Import the

```
#include "/org/smartfrog/services/scripting/groovy/components.sf"
```

This declares three components

`Groovy`	Scripted component using Groovy
`GroovyWorkflow`	The Groovy Component, set to terminate after the start script completes
`GroovyCondition`	Extends the Groovy component with a condition that can be evaluated by other components.

2.1 Attributes

The JavaxScripting, JavaScript and Groovy components all have the same set of attributes

`language`	The scripting language. Use "javascript" for JavaScript, "groovy" for Groovy, etc.
`sfScriptCodeBase`	Prefix to use for resources. Example `"/org/smartfrog/test/system/services/scripting/groovy/"`
`sfDeployResource`	The name of a resource for the deploy-time script
`sfDeployCode`	Inline code to execute at deploy time
`sfStartResource`	The name of a resource for the start-time script
`sfStartCode`	Inline code to execute at start time
`sfPingResource`	The name of a resource for a script to run during liveness checks
`sfPingCode`	Inline code to execute during liveness checks
`sfTerminateWithResource`	The name of a resource for the termination-time script
`sfTerminateWithCode`	Inline code to execute at termination time

Scripted conditions have some more attributes

`sfConditionResource`	The name of a resource for the condition script
`sfConditionCode`	Inline code to execute during condition evaluation
`condition`	The initial (and final) value of the condition variable

All these are in addition to the basic Prim and Workflow attributes, such as `sfClass` and `sfShouldTerminate`. Those attributes are documented in the core documentation.

3 Examples

Here are some examples from the test cases of the components.

3.1 Example: a workflow component

```
LifecycleInlineWorkflow extends GroovyWorkflow {

  sfDeployCode ##
   log.info "deploy"
   #;

  sfStartCode  ##
   log.info "start"
   #;

  sfPingCode  ##
   log.info 'ping'
   #;

  sfTerminateWithCode "log.info 'terminated'";
}
```

This component will print out different messages to the log during the component's lifecycle. Being a workflow component, it will terminate itself after the start message is printed.

3.2 Example: a component that throws an exception

Here is a component declaration that triggers throwing an exception

```
FailOnStartup extends Groovy {

  sfStartCode  ##
   self.fail 'wake up, time to die'
  #;
}
```

It references the `self` variable, which points back to the class that implements the scripting support, and then invokes its `fail(String)` method to throw a `SmartFrogException`.

3.3 Example: a condition

The next example is a condition; a component that implements the `Condition` interface which provides an `evaluate()` method. Container components, such as `WaitFor` and `EvaluateCondition` use the state of these runtime-determined values to choose which component to deploy next.

To pass a return value back to SmartFrog, the script is required to set the `self.condition` value to true or false, depending upon the desired outcome,

```
ConditionInline extends GroovyCondition {
  value true;
  sfConditionCode "self.condition = value";
}
```

This condition does that, by passing in the `value` attribute of the deployed component. This shows another feature of the scripting components, *every SmartFrog attribute of a component is passed down as a variable.*

The attribute copying takes place before any deploy-time script is called; the list is not updated.

3.4 Example: resource file scripts

Inline scripts are simple for short code, but external resource files are a cleaner place to keep complex source files, not least because IDEs now have Groovy and JavaScript support. To use scripts in resource files

1. Create the scripts in separate resource files
2. Point the component at them, as documented below
3. Make sure the build process copies over the source files, otherwise the execution will fail with an error

To point the component at the resource script

1. Optionally, set `sfScriptCodeBase` to the package containing the scripts, such as `"/org/smartfrog/test/system/services/scripting/groovy/"`
   Note the trailing / -this is required.
2. Set whichever of the `sfDeployResource`, `sfPingResource, sfTerminateWithResource,` and `sfConditionResource` to the name of a resource containing a script file. If the `sfScriptCodeBase` attribute is set, the final resource path has that value prepended to it; if omitted, full resource paths must be used.

You can mix inline and scripted resources for different lifecycle scripts. If both a script resource and inline code are provided for a single action, the resource takes priority, unless it is set to an empty string (which is the default).

Here is the example component

```
HelloOnStartup extends Groovy {
  sfScriptCodeBase "/org/smartfrog/test/system/services/scripting/groovy/";
  sfStartResource "hello.groovy";
}
```

It is supported by a file, hello.groovy:

```
log.info "hello from Groovy"
```

The `log` object is the SmartFrog log for the component; it will log messages at info, debug, warn and error, as usual -only the syntax has changed.

4 Implementation

The underlying implementation uses the java6 javax.script scripting engine; the components are designed to work with any of the supported scripting engines, though only a subset have tests.

JavaScript support is built into the JRE, so no extra JAR files are needed. For Groovy, the groovy-minimal-all and groovy-engine JARs are required.

The current release of the components require sf-scripting to be on the classpath; this will change in future

The implementation classes instantiate a script engine for every instance of a scripted component, and set it up with the scoped attributes. They export themselves as the `self` variable, and the log as `log`.

On deployment, startup, ping and termination, any scripts or referenced resources are executed *synchronously.*

The scripted condition component also executes the condition script whenever its evaluate() method is called. This script should set `self.condition` to true for a successful evaluation; `self.condition` to false if the condition is to evaluate false. This allows the scripts to avoid having any return values, which are not always available.

The scripts are not currently compiled. It may be worthwhile supporting compilation, where available, for the condition and ping scripts, as these are likely to be invoked the most frequently during the lifespan of a component.

4.1 Thread Safety

We don't do anything for thread safety; the basic lifecycle methods are synchronized, as is the `Condition.evaluate()` implementation

Engines are allowed to declare their thread isolation policy: multithreaded, thread-isolated or stateless; we do not check or use this information.

5 Exception Throwing and Handling

Some Scripting engines -such as the Rhino JavaScript engine, handle nested exceptions by losing all the data. This is not considered helpful.

5.1 JavaScript

Here is the stack trace from a Javascript component whose startup script was

```
sfStartCode  ##
   self.fail("wake up, time to die");
  #;
```

This throws a `SmartFrogException`, which, if passed through directly, would have been recovered. Instead, only the text message is retained.

```
Caused by: javax.script.ScriptException: sun.org.mozilla.javascript.internal.WrappedException:
 Wrapped SmartFrogException:: wake up, time to die, SmartFrog 3.17.005dev (2009-02-13 11:47:21
GMT)
 (<Unknown source>#2) in <Unknown source> at line number 2
at com.sun.script.javascript.RhinoScriptEngine.eval(RhinoScriptEngine.java:110)
at com.sun.script.javascript.RhinoScriptEngine.eval(RhinoScriptEngine.java:124)
at javax.script.AbstractScriptEngine.eval(AbstractScriptEngine.java:247)
at org.smartfrog.services.scripting.javax.ScriptHelper$LoadedEngine.eval
  (ScriptHelper.java:162)
at org.smartfrog.services.scripting.javax.ScriptHelper$LoadedEngine.resolveAndEvaluate
  (ScriptHelper.java:237)
at org.smartfrog.services.scripting.javax.JavaxScriptingImpl.resolveAndEvaluate
  (JavaxScriptingImpl.java:136)
at org.smartfrog.services.scripting.javax.JavaxScriptingImpl.sfStart
  (JavaxScriptingImpl.java:91)
at org.smartfrog.services.assertions.TestCompoundImpl.sfStart(TestCompoundImpl.java:248)
```

5.2 Groovy

The same script, in groovy, will retain the exception

```
[sf-system-test-junit] Caused by: SmartFrogException:: wake up, time to die, SmartFrog
3.17.005dev (2009-02-13 11:47:21 GMT)
at org.smartfrog.services.scripting.javax.JavaxScriptingImpl.fail(JavaxScriptingImpl.java:182)
at org.codehaus.groovy.reflection.CachedMethod.invoke(CachedMethod.java:86)
at groovy.lang.MetaMethod.doMethodInvoke(MetaMethod.java:230)
at groovy.lang.MetaClassImpl.invokeMethod(MetaClassImpl.java:912)
at groovy.lang.MetaClassImpl.invokeMethod(MetaClassImpl.java:756)
at org.codehaus.groovy.runtime.InvokerHelper.invokePojoMethod(InvokerHelper.java:766)
at org.codehaus.groovy.runtime.InvokerHelper.invokeMethod(InvokerHelper.java:754)
at org.codehaus.groovy.runtime.ScriptBytecodeAdapter.invokeMethodN
   (ScriptBytecodeAdapter.java:170)
at Script11.run(Script11.groovy:2)
at com.sun.script.groovy.GroovyScriptEngine.eval(GroovyScriptEngine.java:243)
at com.sun.script.groovy.GroovyScriptEngine.eval(GroovyScriptEngine.java:81)
at javax.script.AbstractScriptEngine.eval(AbstractScriptEngine.java:247)
at org.smartfrog.services.scripting.javax.ScriptHelper$LoadedEngine.eval
   (ScriptHelper.java:162)
at org.smartfrog.services.scripting.javax.ScriptHelper$LoadedEngine.resolveAndEvaluate
   (ScriptHelper.java:237)
at org.smartfrog.services.scripting.javax.JavaxScriptingImpl.resolveAndEvaluate
   (JavaxScriptingImpl.java:136)
at org.smartfrog.services.scripting.javax.JavaxScriptingImpl.sfStart
   (JavaxScriptingImpl.java:91)
at org.smartfrog.services.assertions.TestCompoundImpl.sfStart(TestCompoundImpl.java:248)
```

Because Groovy preserves exceptions and stack traces, it is more useful for debugging. However filename and location information is not preserved accurately, so determining the exact part of the script which is causing exceptions to be raised may still be difficult.

6 Future Development

The scripted components provide the foundation for simple scripted components.

There are many ways they could be improved, with the underlying goal of making it easier to write SmartFrog components in scripted languages, especially Groovy.

1. Include an sf-groovy RPM.
   This will happen one the move to Java 6 is complete.
2. Allow dot-notation access to attributes.
   This may be possible, through the use of custom binding objects. We would have to ensure that every reference that resolve to a Prim or ComponentDescription returned something which was also a Binding.
3. Asynchronous startup scripts.
   This may become important as more complex scripts are developed.
4. Allow workflow container logic to be implemented in scripts.
   Currently our workflow sequences are implemented in Java; if a complex lifecycle cannot be implemented through aggregating existing workflows, new classes are needed. Allowing workflows to be implemented in script would eliminate this. However, it would be better if we could use the declarative orchestration engine for such workflows, as it is declarative and hence possible to reason over.
5. Make the scripts stateful.
   The scripts should be able to write new variables which would be preserved over the multiple executions of a single component.
6. OSGi Integration.
   Each component creates a new ScriptEngineManager when deployed; this can be created with a ClassLoader as a parameter. We should pass down the current ClassLoader for OSGi classloading to work.
7. Take an array of commands.
   Rather than just a multiline string. There is some support in the APIs for this.
8. Modify the ScriptContext with our own implementation
   This would let us hook up the output and error Reader/Writer interfaces to the SF logger, so scripts that print would have their output routed appropriately.
9. Move from simple scripts to invocable methods; maybe provide some base classes subclassing.
   The aim here is really make it possible to build components from scripts.