SmartScript User's Guide

An introduction to using SmartFrog and BeanShell

SmartScript is the SmartFrog wrapper for the BeanShell scripting environment. BeanShell allows the user to dynamically interpret java source, and also to set up references to any object currently running in one's environment.

BeanShell itself (a complete scripting environment with documentation) is available and fully described at http://www.beanshell.org/ .

We have provided a SmartFrog wrapper for BeanShell, which provides access to a BeanShell interpreter object, and allows manipulation of components on the fly, at runtime. It's eminently suitable for testing or experimental purposes, limited only by your imagination, and by the fact that this component does not work with SmartFrog security *on* (BeanShell uses its own java classloader, which is incompatible with that required by security).

This document is a brief introduction to the use of the SmartScript component, and describes how to include it in your descriptions. We'll start with the archetypal *'Hello World'* example, and we'll show how to interact with the SmartFrog description. Secondly, we'll provide an overview of the BeanShell utilities available through the ScriptPrim component.

A – Bonjour le monde!

Let's first build a simple example that prints a line on the screen.

Here is the raw component as found in the SmartScript package of the SmartFrog release ("scriptprim.sf")

```
ScriptPrim extends Prim {
        sfClass "org.smartfrog.services.scripting.ScriptPrimImpl";
        attributesAsVariables false;            // set to true if you want all attributes of
                                                // the component to be registered in the bsh interpreter
        //port "1234";                          // create a http daemon on the specified port
        //sfCodeBase "http://localhost/"                // prefix for location of scripts
        //sfDeployCode "";          // code executed during sfDeploy
        //sfStartCode "";                       // code executed during sfStart
        //sfTerminateWithCode ""; // code executed during sfTerminateWith
}
```

We'll start by executing a System.out.println() instruction during the deploy phase. All we have to do is replace the sfDeployCode value by the actual string representing the code we want the BeanShell interpreter to execute.

The following description does exactly that. Copy-and-paste the framed text beneath into a new file (we'll call it *tutorial.sf*).

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

sfConfig extends Compound {
        simpleScript extends ScriptPrim {
                sfDeployCode "System.out.println( \"Hello World!\");"; //opening brackets have to be
        escaped
        }
}
```

To deploy it, use sfStart or the sfGui as described in the SmartFrog user manual.

Now if we want to write more than one line of code, and to avoid having to escape any special character, SmartFrog does provide us with a multi-line tag.
Let's add to the previous component a loop to be executed during the sfStart component lifecycle phase

```
sfConfig extends Compound {
        simpleScript extends ScriptPrim {
                sfDeployCode "System.out.println( \"Hello World!\");";
                sfStartCode
                        ##
                        for (int i = 0 ; i < 10 ; i++) {
                                System.out.println(" I can count " + i);
                        }
                        #;
        }
}
```

Note:

Do not forget the **;** after the ## ……# . The SmartFrog parser needs it to recognize the end of the line.

B – Playing with components

Let's now spice things up a little by actually manipulating the SmartFrog components.

The BeanShell interpreter allows you to *bind* an object to a reference[1]. In the SmartFrog case we bind the actual SmartScript component deployed by SmartFrog to the 'prim' name.

The *prim* keyword

In the code or script you write in the description, the keyword ***prim*** gives you access to the ScriptPrim component itself. And to its complete API…

Let's take the previous component and make it display its own SmartFrog reference during the start phase. The API function is *sfCompleteName()*, and it returns a *Reference* object that we'll turn into a String.

```
sfConfig extends Compound {
        simpleScript extends ScriptPrim {
                sfStartCode
                        ##
                        String myName = prim.sfCompleteName().toString();
                        System.out.println(" I am " + myName);
                        #;
        }
}
```

Interestingly, the 'myName' String variable may be set during the deploy phase and displayed in a later phase:

```
sfConfig extends Compound {
        simpleScript extends ScriptPrim {
                sfDeployCode "String myName = prim.sfCompleteName().toString();";
                sfStartCode "System.out.println(" I am " + myName); ";
        }
}
```

The 'attributesAsVariables' attribute

In the 'scriptprim.sf' description of the base script component, is the 'attributesAsVariables' attribute, which we haven't yet used. If set to true, this component binds *each* attribute of the SmartFrog component to objects in the interpreter, using the name of the attribute itself. Thus any attribute can be used in the scripts by just calling its name.

For example:

[1]The actual details of this are in the code, but to make an object available in an interpreter, the code is more or less : Interpreter.set ("name",value);

```
sfConfig extends Compound {
        simpleScript extends ScriptPrim {
                attributesAsVariables true;
                myName "Bunny Rabbit";
                sfStartCode "System.out.println(\" I am \" + myName); ";
        }
}
```

Here's what happened:

The 'simpleScript' component reads all of its attributes, and creates a BeanShell interpreter as usual. It looks up the value of the 'attributesAsVariables' attribute, and as it is true, enumerates over the whole of its SmartFrog context, binding in the bsh interpreter every attribute name to its value.

In this case the value "Bunny Rabbit" of the attribute called 'myName' as been bound to a String 'myName' in the interpreter[2]. The interpreter can then execute the code in the sfStart phase, replacing the myName String by its value.

Let's take another example. If 'attributesAsVariables' is set to true, any link contained in the ScriptPrim component's context can also be used. Let's create a reference to another component in the description, and then use it to read an attribute in this component.

```
sfConfig extends Compound {
        anotherComponent extends Compound {
                verySecretAttribute 1234;
        }
        simpleScript extends ScriptPrim {
                attributesAsVariables true;
                dummy LAZY ATTRIB anotherComponent;
                // the reference to 'anotherComponent' is now bound in the interpreter.
                sfStartCode
                        ##
                        // dummy is a reference, let's first resolve it as we would in SmartFrog
                        org.smartfrog.sfcore.compound.Compound c =
                        (org.smartfrog.sfcore.compound.Compound) prim.sfResolve(dummy);
                        // then use the compound.
                        int magicNumber = c.sfResolve("verySecretAttribute");
                        System.out.println(" I have found " + magicNumber);
                        #;
        }
}
```

Note:
You may have found the line:

org.smartfrog.sfcore.compound.Compound c = (org.smartfrog.sfcore.compound.Compound) prim.sfResolve(dummy);

[2] interpreter.set('myName',this.sfContext().get('myName'));

quite heavy. It is. BeanShell needs the complete class name to be able to load it (remember the bsh interpreter uses its own classloader).

Fortunately, BeanShell uses reflection, and that allows a much simpler syntax. Try replacing the sfStartCode by:

```
sfStartCode
##
        int magicNumber = prim.sfResolve(dummy).sfResolve("verySecretAttribute");
        System.out.println(" I have found " + magicNumber);
#;
```

Another solution is to *import* the class before using it:

```
sfStartCode
##
        import org.smartfrog.sfcore.compound.Compound.*;
        Compound c = (Compound) prim.sfResolve(dummy);
        int magicNumber = c.sfResolve("verySecretAttribute");
        System.out.println(" I have found " + magicNumber);
#;
```

C – BeanShell Utilities: Servers and Scripts.

BeanShell comes with a number of useful services which we can benefit from, including a server and the ability to execute scripts from files or URLs.

Starting a bsh server

BeanShell allows you to deploy a server to make remote call to an interpreter. The BeanShell command itself is:

```
server(portNumber);
```

If the server has been deployed on *host1*, then connecting to this server can be done through http at the following address:

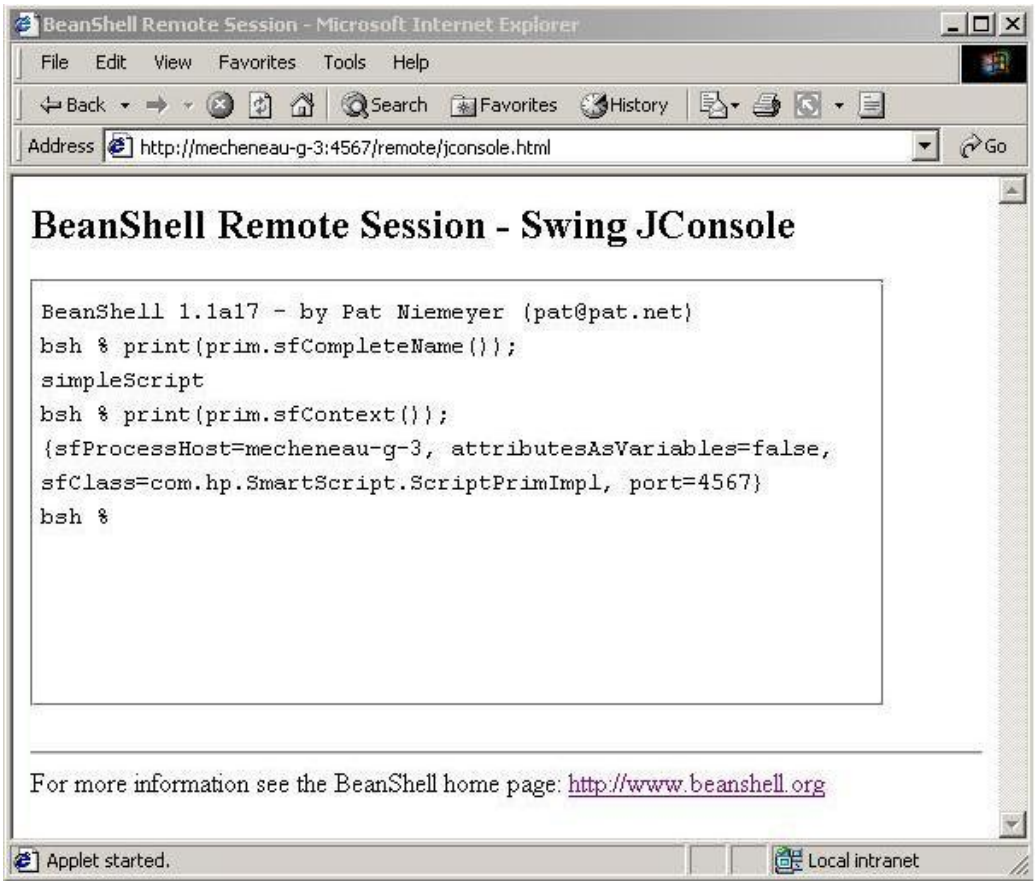
http://host1:portNumber/remote/jconsole.html

In the component itself, just specify a *port* attribute and a server will be deployed on the host where the component itself was deployed.

Try deploying the following example:

```
sfConfig extends Compound {
        hostname "yourhostnamehere";
        simpleScript extends ScriptPrim {
                sfProcessHost ATTRIB hostname;
                port 4567;
        }
}
```

Of course, the *prim* can be used from here, as can be seen in the following screenshot:



Using script files

A BeanShell interpreter can read '.bsh' files and interprets them as scripts.

Once again it's pretty straightforward: if any of the sfDeployCode, sfStartCode or sfTerminateCode attributes is a valid URL, the corresponding script is fetched from the webserver and handed off to the BeanShell interpreter.

As an example, let's create a minimal BeanShell script that just launches a desktop console. Save the following lines in a file named '*tutorial.bsh*':

```
print(" I come from a file ");
desktop();
print(" I launched a console ");
```

```
sfConfig extends Compound {
        fileName "completeFilePath/tutorial.bsh"; //e.g. "file:///C:/SmartFrog/examples/tutorial.bsh".
        // n.b.: any valid URL would work
        simpleScript extends ScriptPrim {
                sfDeployCode ATTRIB fileName;
        }
}
```

You may specify the sfCodeBase attribute in the ScriptPrim component to be used as a prefix for

any file name.

For example:

```
sfConfig extends Compound {
        simpleScript extends ScriptPrim {
                sfCodeBase "http://myhost:4355/SmartFrog/scripts/";
                sfDeployCode "script1.bsh";
                sfDeployCode "script2.bsh";
        }
}
```

is strictly equivalent to :

```
sfConfig extends Compound {
        simpleScript extends ScriptPrim {
                sfDeployCode "http://myhost:4355/SmartFrog/scripts/script1.bsh";
                sfDeployCode "http://myhost:4355/SmartFrog/scripts/script2.bsh";
        }
}
```

If the script comes from a local file, you can also build the complete filename from the description and hand it off to the interpreter directly.

```
sfConfig extends Compound {
        fileName "tutorial.bsh";
        urlPath "C:/CVS/SmartFrog/src/org/smartfrog/services/scripting/"; //no 'file:///' here…
        simpleScript extends ScriptPrim {
                attributesAsVariables true;
                fileName PARENT:ATTRIB fileName;
                urlPath PARENT:ATTRIB urlPath;
                sfDeployCode "completeFileURL = urlPath+fileName; ";
                sfStartCode "this.interpreter.source(completeFileURL);";
        }
}
```

D – The 'expression' SmartScript function

Functions in SmartFrog allow the user to modify the description itself before deployment (see 'The SmartFrog manual' section 5.7 for more details.). For example you can build a URL by concatenating several strings, or get the current date, or a random number as value for an attribute, etc. Functions are description themselves (and as such may be extended to suit your needs), whose parameters are given as attributes. Script functions are also provided; you can therefore provide an expression that will be evaluated before deployment. A few examples should make things clearer.

Here is the template for the 'expression' function :

```
expression extends {

        phase.function "org.smartfrog.services.scripting.Expression";

        // myvariable myvalue

        // exp "function(myvariable)" ...

}
```

Let us, for example, extend the expression function to get a generic binary sum operator.

```
#include "org/smartfrog/components.sf"
#include "org/smartfrog/services/scripting/function.sf"          // include the function template

value1 5;
value2 6;
value3 "Hel";
value4 "lo";


// Extend the template
sum extends expression {
        a 0;
        b 0;
        exp "a + b";            // will be replaced after parsing by the value of a+b (0 by default)
}

// and use it in two different ways:
sfConfig extends Compound {
        numericSum extends sum {
                a ATTRIB value1;
                b ATTRIB value2;
        }
        stringSum extends sum {
                a ATTRIB value3;
                b ATTRIB value4;
        }
}
```

The previous sfConfig, once parsed, is turned into:

```
sfConfig extends Compound {
        numericSum 11;          // a + b was value1 + value2 = 5 + 6
        stringSum "Hello";      // a + b was value3 + value4 = "Hel" + "lo"
}
```

In this example of course 'a' and 'b' have to be of coherent types, otherwise beanshell (i.e. Java) will not be able to compute a sum.

More generally, the content of the 'exp' attribute may be any piece of Java code you wish; the final return value of this code will be the value of the function. Let us build a factorial example (probably highly useless but perfect as a short demonstration).

```
#include "org/smartfrog/components.sf"
#include "org/smartfrog/services/scripting/function.sf"          // include the function template

// Extend the expression template
factorial extends expression {
        // n …                  // the number whose factorial we want to compute
        exp ##
                int c = 1;                  // will be our return value
                for (int i = 1; i <= n ; i++){
                        c = i * c ;
                }
                return c;                   // the actual valut of the function
        #;
}

// and use it in a description:
sfConfig extends Compound {
        myNum extends factorial {
                n 5;
        }
}
```

This sfConfig, once parsed, is turned into:

```
sfConfig extends Compound {
        myNum 120;              // 5!
}
```

Another use of this could be to actual use the script function to pull a value out of a web server for example, or to compute well-formatted unique Ids, etc.

Other examples of the 'expression' SmartFrog function may be found in the SmartFrog release, in the file: com/hp/sfExamples/functions/function_expression.sf

For more details on how to build your own functions in SmartFrog, you can see the advanced section 17.1 in 'The SmartFrog manual'.

E – A Few Comments

You will find four other examples available in the SmartFrog release, in the org/smartfrog/services/scripting/examples/examples.sf file.

We'll conclude by highlighting problems that may arise from the deployment order of components, and classloader conflicts when SmartFrog security is turned on.

Deployment Order

Bear in mind that if the SmartScript component is deployed in a normal Compound, the sfStartCode is executed after the sfDeployCode. The Compound object calls the sfDeploy() methods of all its children, and then successively calls their sfStart() methods.

Interaction between components may become complex at this stage.Try to deploy the following example:

```
sfConfig extends Compound {
        firstScript extends ScriptPrim {
                verySecretAttribute 1234;
                sfDeployCode
                        ##
                        String myName = prim.sfCompleteName().toString();
                        System.out.println(myName+": My secret attribute is
                        "+prim.sfResolve("verySecretAttribute")) ;
                        #;
                sfStartCode
                        ##
                        System.out.println(myName+": My secret attribute is now
                        "+prim.sfResolve("verySecretAttribute"));
                        #;
        }
        secondScript extends ScriptPrim {
                attributesAsVariables true;
                theOtherScriptRef LAZY ATTRIB firstScript;  // a reference to the first script
                sfDeployCode
                        ##
                        String myName = prim.sfCompleteName.toString();
                        p = prim.sfResolve(theOtherScriptRef);  // a pointer to the first script component.
                        int magicNumber = p.sfResolve("verySecretAttribute");
                        System.out.println(myName+": I have found " + magicNumber);
                        magicNumber = 4321;
                        System.out.println(myName+": I do not like it. I prefer " + magicNumber);
                        p.sfResolveAttribute("verySecretAttribute",magicNumber);
                        #;
        }
}
```

As you can see, this might lead to some confusion, as the sfStartCode is executed *last*.

ScriptPrim termination issues: the BeanShell server

The BeanShell server can't be stopped. Consequently a ScriptPrim component that has started a server (for example with the 'port' attribute set) can't satisfy the requirements of the Termination lifecycle phase, as the server remains listening on the port.

If need be, one solution is to start the ScriptPrim component in a separate process (see the *sfReference* document for the different ways to do so), and to terminate the whole process to effectively wipe out the BeanShell server.

Security (absence of)

BeanShell cannot be used with SmartFrog security turned on. BeanShell renders your application widely accessible, both by providing complete access to the components themselves via any included SmartScript component, and by allowing this to be done remotely thanks to the BeanShell server. This means that BeanShell is not currently suitable for use where a secure environment is required.