Guide to new features in SmartFrog

Olivier Pernet and Julio Guijarro - August 2007

The present (hopefully) short guide explains how to use the new features introduced in SmartFrog during the summer of 2007. These are mostly about clearly separating user-provided code from the core of SmartFrog, and giving users means to declare what code they want to use in a deployment. That information used to be largely implicit, but is now explicit and thus easier to control.
This entails changes to the way systems are deployed, which will be detailed here.
As part of this work, integration with the OSGi Service Platform has been introduced, and SmartFrog users can now take advantage of the OSGi platform features if they want to. Backwards compatibility is still preserved and the OSGi features are completely optional.
The source code this document relates to is available in the SmartFrog Subversion repository at:
https://smartfrog.svn.sourceforge.net/svnroot/smartfrog/branches/core-branch-osgi

Daemon

The daemon should now require much less care, as no configuration changes are needed to deploy an application on any given daemon. This means no classpath/lib directory changes, and no codebase changes. This should also help deploy several different applications on the same daemon.

To take advantage of the new OSGi-related features, the daemon needs to be deployed in an OSGi environment. This is described in [Status of SmartFrog OSGi integration].

SmartFrog Descriptions

Amongst the new features introduced is the concept of an application environment. An application environment contains all components that need to be pre-deployed before your application can be.

This notably includes components implementing the new ClassLoadingEnvironment interface. This interface encapsulates access to code. We currently have an implementation that fetches code from a network location: URLClassLoadingEnvironment, and one that loads it from an OSGi bundle: BundleClassLoadingEnvironment (it also installs the bundle in the OSGi framework when deployed).

The class loading environments are not the only components that can be put in application environments. If you've been writing custom deployers that are packaged with the rest of your application classes, you'll need to declare them there, too. This implies that deployers are now full-fledged SmartFrog components see section Framework extension points for details. In fact, all components described in this section will need to be declared in the application environment.
If you're only using the deployers that come with SmartFrog though, you don't need to do anything special about them.

Now a distinction needs to be made between the application environment your application needs at parse time, and the one it needs at deployment time. It's easy to see, for example, that custom deployers are not going to be any use for the parser; conversely, your SmartFrog descriptions (.sf files) are only needed by the parser. So, the application environment description needs to have separate sections for those. Here's an example of such a description:

```
#include "org/smartfrog/environment.sf"

sfConfig extends ApplicationEnvironment {
        parser extends ParserEnvironment {
                sfProcessComponentName "envParserAppFoo";
                clEnv extends URLClassLoadingEnvironment {
                        codebase "http://host2/descriptions.jar";
                }
        }

        target extends TargetEnvironment {
                sfProcessComponentName "envTargetAppFoo";
                clEnv extends extends URLClassLoadingEnvironment {
                        location [ "http://host1/components.jar",
                                "http://host1/resources.jar" ];
                }
        }
}
```

You can see several things here. First, we're including the new `org/smartfrog/environment.sf` file, which has the basic components you'll need to describe environments. It does not include OSGi-specific components, which you can include from `org/smartfrog/osgi/components.sf`.

Here we're loading code directly from JAR files on the network, by means of `URLClassLoadingEnvironment`. They will need to be signed if security is enabled, as before. We're also using special components: `ApplicationEnvironment`, `ParserEnvironment` and `TargetEnvironment`. You need to use those so that everything works, or you can extend them to suit special needs.

Then, we're using the `sfProcessComponentName` attribute. This is because each environment will be deployed just like a normal SmartFrog application, and thus needs a name that you will use to refer to it. If you're deploying several applications with different environments on the same target, you'll want to make sure that each environment has a different name, to prevent conflicts.

Let's now have a look at application descriptions. All existing descriptions that only use components from the SmartFrog distribution should work unchanged. Some changes are in order though if you're using your own components. Let's have a look at an example that uses the environment defined in the previous example:

```
#include "org/smartfrog/components.sf"
#from envParserAppFoo:clEnv #include "com/foo/components.sf"

sfConfig extends Compound {
        sfClassLoadingEnvironment LAZY PROCESS:envTargetAppFoo:clEnv;
        a extends A {
            quux 42;
        }

        b extends B {
            aRef a;
        }
}
```

Notice here the new parser directive : `#from`. This is used to tell the parser in which `ClassLoadingEnvironment` it will find your included description. If omitted, it will look for it in the SmartFrog distribution.

The new `sfClassLoadingEnvironment` attribute is used like the now-deprecated `sfCodeBase` attribute. Its value gets propagated down the tree so you don't have to repeat it. The framework will use the attribute to select where to load your component class from, thus you can get errors mentioning a `ClassNotFoundException` if you point it to the wrong component. Also, you need to make sure that you're using a reference that does not hard-codes the host and process name. This is because your application environment gets replicated on every process automatically, as class loading can only happen within a single process. So the reference needs to point to the replicated version, and not the parent one.

Important note: If you've been using the `sfCodeBase` attribute for similar purposes, we intend to have backwards compatibility for it for a while, but you should be moving to using application environments as a more flexible alternative.

Starting a SmartFrog application

The `sfStart` command has changed quite a bit, hopefully for the better. It can now deploy an arbitrary number of environments before your actual application. Here's its abridged syntax:

```
sfStart -t target.host
        [-e env.sf [env2.sf...]]
        [-d AppName app.sf [-d AppName2 app2.sf...]]
```

And here's a typical usage example :

```
sfStart -t host -e env.sf -d MyApp myapp.sf
```

An exemple that deploys an application only using components from the SmartFrog distribution:

```
sfStart -t host -d MyApp myapp.sf
```

Another exemple that deploys an environment and then several applications that use it:

```
sfStart -t host -e env.sf -d App1 app1.sf -d App2 app2.sf -d App3 app3.sf
```

When providing implementations of extension points (see below), you'll need to deploy two (or more) environments for your application : a first one that gives access to your extension classes, and then one that deploys instances of those classes. Here's the example :

```
sfStart -t host -e env1.sf env2.sf -d MyApp myapp.sf

// env1.sf
#include “org/smartfrog/environment.sf”
sfConfig extends ApplicationEnvironment {
        parser extends ParserEnvironment {
                sfProcessComponentName “parserEnvMyApp”;
                cl extends URLClassLoadingEnvironment {
                        location [ “http://host1/components.jar”,
                                “http://host1/extensions.jar” ];
                }
        }

        target extends TargetEnvironment {
                sfProcessComponentName “targetEnvDeployerMyApp”;
                cl extends URLClassLoadingEnvironment {
                        location [ “http://host1/components.jar”,
                                “http://host1/extensions.jar” ];
                }
        }
}
// env2.sf
#from parserEnvMyApp:cl #include “com/foo/mydeployer.sf”
sfConfig extends ApplicationEnvironment {
        target extends TargetEnvironment {
                sfProcessComponentName “targetEnvDeployerMyApp”;
                myDeployer extends MyDeployer {
                        sfClassLoadingEnvironment LAZY targetEnvClMyApp:cl;
                        // Configure your deployer here...
                }
        }
}

// myapp.sf
#from parserEnvMyApp:cl #include “com/foo/components.sf”
sfConfig extends Compound {
        a extends A {
                sfDeployer LAZY targetEnvDeployerMyApp:myDeployer;
        }
}
```

You can also deploy an environment separately, and later deploy applications that use it. The applications will need to define the `sfApplicationEnvironment` attribute explicitely, as it cannot be guessed by the `sfStart` command in this case.

```
sfStart -t host -e env.sf
sfStart -t host -d MyApp myapp.sf
```

where env.sf and myapp.sf look like this:

```
//env.sf
sfConfig extends ApplicationEnvironment {
        parser extends ParserEnvironment { // ... }
        target extends TargetEnvironment { sfProcessComponentName
envMyApp; // ... }
}

// myapp.sf
sfConfig extends Compound {
        sfApplicationEnvironment LAZY PROCESS:envMyApp;
        // ...
}
```

Finally, note that wherever the script expects a description file name, you can use a complete URL. This allows you to load descriptions from a remote host, or from within a JAR file by using `jar:` URLs. The latter is the only kind of URL you'll be allowed to use when security is enabled, as only resources loaded from a JAR file have an attached signature. Here's an example using JAR URLs:

```
sfStart -t host \
           -e 'jar:http://host2/descriptions.jar!/com/foo/env.sf' \
           -d MyApp 'jar:http://host2/descriptions.jar!/com/foo/myapp.sf'
```

Writing SmartFrog components

The changes are largely transparent here. However, those components that provide services to other components and need to load classes or files to provide this service will need to be looked at.
This is because one cannot anymore expect to have caller code on the classpath. So, for example, if you were providing an XML parser component, and were using `SFClassLoader.getResourceAsStream(userfile)` to load client-provided files, this won't work anymore. Similarly, if you somehow expected to be able to load classes by using `SFClassLoader.forName(userclassname)` when you are not providing the class `userclassname` yourself, you will need to change a few things.

Fortunately for you, there's a new feature that solves this problem. Each component now comes with a `sfClassLoadingEnvironment` attribute that is a reference to the environment a component comes from. By resolving this attribute you get an object implementing the ClassLoadingEnvironment interface, which you can now use to load classes or resources. If you're only dealing with resources, there's a helper class you want to have a look at: `ResourceLoader`, in `org.smartfrog.sfcore.common`. Alternatively, `ComponentHelper` in the same package will take care of both class loading and resource loading.

Framework extension points

The rationale behind most of the changes described here has been to clearly separate code that can be provided by users from the core SmartFrog classes, and as much as possible, turn that code into full-fledged components that can be described and managed by means of the SmartFrog language.

Here's the list of interfaces you can implement to extend framework functionality:

- ComponentDeployer
- PrimFactory
- ClassLoadingEnvironment
- Function
- PhaseAction

The first three interfaces need to be implemented by SmartFrog components; if you want to use one, you will need to declare it in your application environment. The latter two have not been converted this way to keep things simple.

Note: The RootLocator interface used to be implementable by users as well; all its implementations now need to be packaged with the SmartFrog core classes.

Deployers

If you've been using custom deployers already, you'll know that these didn't need to be Prims before. But if you're extending a deployer from the SmartFrog core this will be taken care of for you. You'll still need to make some changes. First, you need to recompile your deployer, and change its constructor, as the deployers in the SmartFrog core have switched to a no-argument constructor.

Then, you need to change the descriptions that used it from this :

```
foo extends Foo {
        sfDeployerClass "com.foo.MyDeployer";
}
```

...to this :

```
// in the application environment description
target extends TargetEnvironment {
        sfProcessComponentName "targetEnvFoo";
        myDeployer extends MyDeployer;
}
// in the application description
foo extends Foo {
        sfDeployer LAZY targetEnvFoo:myDeployer;
}
```

PrimFactory

There is a newcomer in the above list that warrants further introduction : the PrimFactory interface. This interface allows you to change how a component is created from a ClassLoadingEnvironment and is description. The default implementation uses the familiar *sfClass* attribute and loads the designated class in the environment. But more sophisticated uses can be though about : doing AOP, retrieving persisted components from a data store, etc.