Writing a SmartFrog component

Last edited: 25 Mar 2004

A SmartFrog component is a Java class or set of classes that host parts of a distributed system under the SmartFrog deployment framework. If the parts of the system are themselves implemented in Java, they may actually be executed in the SmartFrog JVM itself, using the SmartFrog framework as a means of starting and stopping the code. If the program is a native executable, or needs to run in its own process, SmartFrog can instead launch the process, probe it for availability/health, and then shut it down when needed.

In either situation, SmartFrog provides the means to download the Java classes, deploy the components in the user-requested configurations, manage their lifecycle and undeploy them when needed. The framework also links together components across different systems -any component that takes a reference to second component will be given a remote reference if that second component is hosted on a different machine.

It this is this ability to distribute components across machines based on the declarations in a SmartFrog deployment descriptor, and the ability to bind those components together, that gives SmartFrog its power. To take advantage of that power, developers need to

1. Write components that allow SmartFrog to deploy, probe and undeploy their software.
2. Provide configuration options that allow developers and system administrators to configure their software in SmartFrog.
3. Document the configuration options.

This document shows developers how to write a SmartFrog component, covering theory "what is a component" and practise "how to debug a component". It assumes a working knowledge of Java and Ant.

Core Concepts

Classpaths

When SmartFrog starts, it runs in whatever classpath that the JVM was started with. In the batch files, that includes *.jar in SFHOME/lib; while in the ant tasks it includes whatever you declared in the classpath for the relevant task.

Networking

SmartFrog objects are networked, using Java's RMI protocol. It is useful to have some RMI knowledge when writing a new component; the section in the Java1.4 edition of *Learning Java* (O'Reilly Press) is mostly sufficient; *Mastering RMI* by Rickard Öberg . The key points are:

1. RMI offers remote access to interfaces that extend the `java.rmi.Remote` interface; all remoteable interfaces have to do this.
2. They also have to offer the option of throwing a `java.rmi.RemoteException` on every call, as users of the interface need to know about communications errors.
3. Remote objects will not have access to any shared state in the caller application; all communication should be via method calls and serialized Java objects used in the calls, rather than in implicit state sharing behind the scenes.
4. If you pass a remoteable interface back inside an object, then a remote reference will be sent back, referring back to the local object.
5. Distributed garbage collection is still an ongoing research problem.

Every component has to have its own remote interface for any remote manipulation -without that rmic doesn't generate the relevant proxy classes. Either the component declares its own Remote interface, or it must explicitly implement Prim. Having the parent class declare support for Prim is not enough.

They must also run the class through `rmic`, such as via the Ant task `<rmic>`, to generate proxy classes which must be included in the redistributable. Apparently Java1.5 eliminates such work, but then so can SOAP.

Thread Safety

Component methods do need to think about thread safety. The core lifecycle methods are synchronous, so a component will not be stopped in `sfTerminate()` while it is starting inside `sfStart()`. For that reason, incidentally, taking a long time to start in sfStart() is strongly discouraged -that method should start worker threads to perform time consuming actions.

The remote interface of a component can be called by any component in the distributed system that can locate the component, and has knowledge of the custom interface. Assume that any public method in this interface can be called on separate threads, and code accordingly. That means marking methods as `synchronized`, using `synchronized() {}` code blocks, or other techniques to ensure thread safety. Take care not to create deadlocks, of course.

When invoking any component, remember that it may be remote, and even if local, it may execute in a separate thread. To quote the RMI Specification *"The RMI runtime makes no guarantees with respect to mapping remote object invocations to remote threads.*" What that means in practise is this: any call to another component, in a `synchronized`() block of code, may lead to deadlocks.

Shutdown

Although the lifecycle method `sfTerminate()` is where termination begins, components should normally override `sfTerminateWith()` when they want to implement termination code.

Shutting down third-party code running in-process turns out to be surprisingly hard, depending upon how the component was written. The primary problems are usually (a) the component was never written to be shut down, except by killing the JVM or unloading all the classes, or (b) the component cannot be restarted after a shutdown.

In the latter case, one would think that deleting all references to an object and forcing a garbage collection would be sufficient to trigger a cleanup, but any object that creates persistent static data structures has effectively leaked data that can only be fixed by unloading the entire JVM.

If the component is written to be contained, then any failures at shutdown time are defects that need to be corrected. If you are trying to wrap something that was never intended to be embedded, it is harder to persuade the component developers that their code is at fault. This is the same problem the Ant and Tomcat JSP teams have with products such as javac -it used to leak big static data structures, but every defect of that nature filed against Sun used to result in "don't do that then" responses, before it finally got corrected. The solutions these teams have to such leakage is to fork a new JVM, which easier said than done, as there is no `fork()` command in Java; you need to `exec()` a new process with the right settings.

This need to separate startup/shutdown code from the actual execution can actually prove beneficial in a long running system, because the monitoring framework is better isolated from the application itself. An JVM crash does not kill the management code, so the crash is noted and acted on according to the policy specified in the deployment descriptor.

It is critical that the shutdown code is written to shutdown everything it can, even if bits of the shutdown fail. Wrapping operations in exception handlers is the standard tactic here. Exceptions during shutdown should be caught and logged, not rethrown, and every critical operation should be wrapped in its own exception handler, if it is possible that one stage of the shutdown may fail while the next stage could still work.

> *Never try and load resources or dynamic classes (via `Class.forName()`) during termination, as if the source JAR file cannot be loaded, the call will fail.*

It is also important to remember that an object can go straight to shutdown without ever being deployed. For example, if it is instantiated and it (or a child) throws an exception during the `sfDeploy()` call, the `sfTerminate()` method is invoked without `sfStart()` being called, and without `sfDeploy()` actually completing. Throwing an exception during `sfStart()` will also result in `sfTerminate()` being invoked before `sfStart()` finished. What does this mean? It means that you cannot assume that fields which are assigned object references during startup or deployment are actually valid during termination. Attempting to call `server.stop()` on the field `HttpServer server` will result in a `NullPointerException`, if the server variable was not initialised during deployment, because some stage during initialisation -such as resolving the "port" attribute, failed.

> *Always test deployment- or startup- initialized fields for being null in the terminator, and skip the relevant parts of the termination.*

The overall key to a successful and robust termination method is to have everything pre-prepared, so there is no need to resolve references to remote components, load classes or resources, or

Resolving Attributes

References are resolved using `sfResolve()` calls. This call can go over the network, if that is what is needed to resolve something, so the `RemoteException` that is part of the method's signature is there for a reason. It also means that resolution is potentially costly, and that caching of values should be considered.

You can not assume that all references/attributes are in place by the time an `sfStart()` call is made, because that is where run-time attributes may be added to the system.

Adding New Attributes

To add a new attribute to an object, `put()` it on the `sfContext` of a prim instance, usually yourself:

```
sfContext().put("filename", tempFile.toString());
```

Here we are setting the `filename` attribute to the name of a temporary file we have just created.

Remember two things when adding attributes

1. You must, whenever possible, set the attributes in the `sfStart()` method, and not `sfDeploy()`. The convention is that all run-time attributes should be set by the time deployment begins.
2. Although the signature of `put()` is `(Object, Object)`, always use a string for the attribute name, and a serializable/remotable datatype for the value. Simple datatypes work best.

Loading Resources

You should not use `this.getClassloader.loadResource()` to load a resource; instead delegate it to a secure classloader:

```
SFClassLoader.getResourceAsStream()
```

The secure classloader only loads stuff from signed jars, and does it in a way that is paranoid: it copies resources into a buffer before giving the caller an `InputStream` handle. This ensures that nobody else can get at the bits of the resource after validation and subvert them.

You can pass in a normal resource string `"org/smartfrog/data.properties"`, in which case the resource is loaded from the current classpath. Alternatively, provide a full URL, including http:// references.

Inter-Component Communication

Components can talk to each other over RMI. First, they must find each other. Here are some methods:

- Child enumeration: Compound-derived components can iterate through the Vector of children.
- Parent-tree walking: Components can walk up the tree of components (`Prim.getParent()`)
- Reference resolution: Get a reference to a Prim and cast it *to the interface you need*.

It is imperative that you rely on interfaces, and not implementation classes, because remote proxy class have different classes from the implementation classes, yet they offer the same interfaces. Any check for methods should be `implements` of the interface, all casts should be to the interface.

Once a component has located another, it can use direct communication over RMI. This is how the core components in the framework all communicate.

Build Process

The build process is relatively simple; the extras on top of a simple compilation are RMI compilation, copying over any .sf files into the build directory, and signing the JAR files for security.

It is prudent to run the ant task `<sf-parse>` over the SmartFrog files, so that any parse errors get addressed before deployment begins. The parser also displays the final 'resolved' file, which can be examined by hand to verify that everything 'looks good'. Normally deployment is the full, rigorous test, but examining configuration by hand is a good learning exercise, as it shows you how the resolution process really works.

Testing

How do you test a deployed system? Consult the specific documentation on testing under SmartFrog.

Packaging

For security reasons, the component should be built into a jar file, with a manifest, signature and sealing of all classes. Without this, the component cannot be used in a secure SmartFrog system, which means that it cannot be used to safely deploy production code.

Classloading

SmartFrog has a stricter classloader model than normal, because of

1. its ability to securely download code from remote locations
2. the ability of different components to declare new lists of remote URLS in their `sfCodeBase` attributes

The effect is that loading classes is hard, and resources even more so.

Troubleshooting

Parse trouble

Parse trouble should be hit before deployment.

What does not get caught at this time are misspellings of attributes, missing attributes or other semantic errors.

Deployment

Deployment is where the components themselves verify that all their needed items are found; anything missing will result in an error such as

```
 Unresolved Reference: threads, source: HOST chamonix:sfRunProcess:cddlm
```

This lists the reference, and the context, which includes the hostname and all the elements used in the resolution stage.

Illegal ClassType exceptions mean that the type of an attribute is not what the code wanted:

```
SmartFrogResolutionException:: Illegal ClassType, Unresolved Reference: threads, source: HOST
chamonix:sfRunProcess:cddlm
```

Here the line was

```
threads "4";
```

instead of

```
threads 4;
```

This is important. Unlike Ant, which automatically does type conversion depending upon whether the task wants numbers, strings, booleans or file references, SmartFrog needs you to get your type right in the language files. This also forces the implementation to be consistent with its interpretation of attributes; Ant tricks where attributes have multiple meanings are only valid if the type of all meanings is consistent.

Debugging

Debugging anything server-side is always complex. You have multiple options.

Logging

There is a logging mechanism which is built in to SmartFrog. It is designed to be dynamically configurable from deployment descriptors, and can even log to remote systems. Use it as much as you can -it has an API which matches that of Apache `commons-logging`. For efficiency, consider checking for debug level logging before concatenating large strings of debug output.

Assertions

Java 1.4 assertions are an excellent way of adding extra checks into code. As there is no impact of having assertions in when they are disabled (the JVM runtime discards them), there is little cost in adding them to code. The sole dangers are (a) that a developer adds a check that is stricter than necessary, causing unwanted assertion failures, and (b), a developer may add side-effecting code to an assertion. The latter can be avoided by never putting side-effecting code into an assertion check, and by never calling any methods from an assertion statement, even if you "know" that it has no side effects. This is because there is no way to be sure that the method may not become side effecting in future.

Write Java1.4 code with assertion checks in them, and compile them with the source option of the compiler set to 1.4 (or later, for more recent platforms). In Ant, set the attributes `source="1.4"` and `target="1.4"`.

Then run SmartFrog with assertion checking enabled.

```
      <sf-run classpathref="run.classpath"
        logStackTraces="true"
         >
        <assertions enableSystemAssertions="true">
          <enable/>
        </assertions>
        <application name="cddlm" descriptor="/${deploy.sf}"/>
      </sf-run>
```

Attach to a running program

Configure the Java call used to start SmartFrog to enable debugging (JVM dependent, usually `-Xdebug` with options).

Running the program from your debugger.

Configure the debugger to run SmartFrog by setting the classname to run to `org.smartfrog.SFSystem` , the classpath to include `smartfrog.jar` and your own classes.

You then need to set up the appropriate parameters to run the program. A good combination is JVM arguments:

```
-Dorg.smartfrog.logger.logStackTrace=true
-Dorg.smartfrog.sfcore.processcompound.sfProcessName=rootProcess
-esa -ea
```

and then options to `SFSystem` of something like

```
-a cddlm:DEPLOY:/org/smartfrog/services/cddlm/system.sf::localhost:
```

That is the syntax for deploying an application on the local host. That is sufficient to test that deployment works, as you can set breakpoints on your component's `sfDeploy()` method and step through the stages.

Remove the `-e` option and your program runs continually until the program is killed or the daemon shutdown. To shutdown cleanly it is useful to have an ant task for this purpose; a simple target (such as this one from `common.xml`):

```
  <target name="shutdown"
    depends="use-smartfrog-tasks"
    description="shut down a local smartfrog daemon">
    <sf-stopdaemon failonerror="false" />
  </target>
```

This will shut down a daemon if one is running, and let you test your shutdown code, too. If you `<import> core/common.xml` into your build file, you get this target for free.

One surprise when debugging is liveness checks will kick in while you are debugging, and they are prone to fail because you are stepping through the application itself. While these checks are correct -the application is not live, it is being debugged, it is very inconvenient to have this happen. `Set sfLivenessDelay 0;` across your components to avoid this by disabling liveness.

You do not actually need to declare the deployment descriptor on the command line, if you want the daemon to sit and wait for deployment requests. Just make sure the classpath loads all the JAR files that will be needed for debugging, because you need that to set breakpoints on custom components.

Style Guide

Provide your own `Remote` interface

A component must declare that it implements an interface that extends `Remote`. It is sometimes tempting to declare Prim as that interface, and so avoid having to write a new interface. This is a short term tactic.

1. The interface makes a good place to put the names of all attributes that are (or have been) supported.
2. It lets you probe the type of a proxy class by asking if it `implements` an interface.
3. It lets you add methods in future.

Provide a schema

The schema does two things: it permits parse-time validation of the type of attributes, and verifies that mandatory attributes are present. It also provides a cue to the component user as to what attributes are allowed, and their type. The schema can be an essential part of the component documentation.

Place default values of optional components in the descriptor

When resolving optional attributes, the component code can configure what the default values should be. This is placing the defaults into the code -compile time binding. Placing the default values into the component template descriptor not only makes it obvious to the reader of the file what the defaults are, they make it easier to change those defaults.

There is a consequence of this action: what to do if the optional attribute is completely undefined? Well, if the base declaration declares the defaults, why not just make that attribute mandatory?

Do (almost) nothing in the constructor

Although it is conventional to initialise an object instance in the constructor method, you cannot use any of the SmartFrog API or rely on any part of the framework until it is initialised -which does not happen until `sfDeploy()` is called.

Initialise the object in `sfDeploy()`

As the object cannot be initialised in the constructor, the place to do it is in the first method called by the framework. This is `sfDeploy()`. Place the primary initialisation code in this method.

Always call the super methods

When you subclass, `sfDeploy()`, `sfStart()`, `sfTerminate()`, `sfPing()` or another method implemented by the base class, always invoke the parent class's base method first. Many of the parent methods perform essential parts of managing the lifecycle of the SmartFrog component. Even if a parent method does not do anything, *it may do so in a future version*. Always call the parent method, usually as the first action of the overclassed method.

Make `sfTerminate` bullet-proof

This was discussed earlier. The termination code should:

- test for and handle null references
- not load classes or resources
- catch failed operations, and proceed to the next stage in the termination.
- set references to null, during termination. This ensures that regardless of the lifespan of the component instance, any resources used are released.

Do not ignore exceptions, except during termination.

Throw exceptions, or wrap them in SmartFrog exceptions (use `SmartFrogException.forward()`). If you must ignore an exception, log at a very low level the fact that it was ignored.

Design for distribution

Components will run on other hosts from other components. So do not assume that a filesystem is shared, that non-serializable, non-remotable objects can be shared, or that there is anything in common with the remote system other than the RMI channel.

Use attributes to share information

If you need to have two components communicate, there are two means. The classic distributed objects pattern is to declare methods on an interface, and have the caller import the interface, then make method calls. This is very brittle against change, as it only permits the caller to talk to implementations of this interface.

A better tactic in the SmartFrog context is to share information via attributes. If a component sets attributes on itself to publish facts, then anyone interested in the facts can ask for them.

This has the advantage of looser coupling of components; there is no direct call on the component, so much as an indirect query for attributes attached to a component. It permits the caller to extract the information from anything that adds the attributes: *there is no need to bind at compile time to a common interface.*

The disadvantage is the coupling is looser. There is no way to request an operation; only query values. The values must be set by the component at some point in its lifecycle or they are absent -the caller may have to spin awaiting it, and time out if it does not arrive. You are also somewhate limited in the types of information that can be exchanged, though no more than RMI normally enforces.