Ant Tasks for SmartFrog

Introduction

This document covers the Ant tasks for SmartFrog. There is a corresponding presentation and set of automatically generated manual pages for all the tasks. Because the latter are generated directly from the source code, they are the most accurate and up to date

Comparing Ant and SmartFrog

Apache Ant and HP SmartFrog are actually quite similar in interesting ways. Both are frameworks for executing plug-in components with the order and options driven by a configuration file.

Ant

Ant was written to compile Java programs on a single computer. It has a dependency model oriented towards a process of generating files from other files -most core tasks tend are dependency driven and do no work if the destination file(s) are considered up to date.

It is strongly multiplatform, and core datatypes (files, paths, filesets) can be configured in a platform independent manner.

Ant has many extension points -Tasks are the most well known, but listeners, loggers, conditions, mappers and selectors are some of the other ways that it can be extended. And, being open source, it can be expanded in interesting ways.

Although written purely as a replacement for the classic makefile, it has expanded its role to cover simple deployment, primarily as part of a build process. It also finds a use as an application launcher (commons-launcher), and provides the compilation services for the JSP engine in Tomcat 5.x. It has also been used as the workflow engine for GridAnt, a use which involved running a single build with thousands of nodes for days at a time. This final use is way beyond the original scope -Ant has effectively become the de facto workflow language for Java projects. This is despite the fact that it lacks the core features of workflow: a persistent store of ongoing state, and integrated conditions/error handing. Ant's default error handling policy is "halt the build on an error", which is perfectly consistent with its core role as a build tool. Many tasks offer custom error handling, via the `failOnError` flag, which provides the other mainstream policy of "ignore the error"

Ant tasks usually extend the class `org.apache.ant.Task,` although this is not mandatory -by using reflection there is no need to do anything other than offer the method void `execute()`. The Ant runtime uses reflection to determine which attributes and nested elements are supported, and binds fragments of an XML file to the task by calling these methods. Then it calls the `execute()` method to execute the task.

SmartFrog

SmartFrog was written to deploy applications, undeploy them and monitor the state of the applications during their deployment. It also views a large-scale application as composite of multiple smaller applications, and manages them as such.

The default failure model for SmartFrog is that if a part of an application fails, the compound system has failed. Failures propagate up the tree. The default action on failure is to terminate the application. It is possible to provide new compound components that have different policies -this is used to provide workflow constructs such as retry, repeat, ignore, try/catch and the like. It is also possible to manipulate the graph of components -the workflow components provide this.

Although originally written as a subsidiary project, the workflow components have become an integral part of the system. This is because they have proven so useful for describing the steps to deploy and undeploy complex systems.

SmartFrog can be extended in interesting ways. Most unusually, it is possible to extend the multiple stages of processing a document, and so fundamentally change how the language is interpreted -or what language is used as the front end.

The most common extension model is much less ambitious. A *Component* is a Java class that

1. extends `org.smartfrog.sfcore.Prim` or a derivative

2. implements an interface that extends the `Remote` interface of Java RMI
3. implements the lifecycle methods of a SmartFrog component: `sfDeploy()`, `sfStart()`, `sfLivenessCheck()` and `sfTerminate()`

It is essential that components are remotable. The SmartFrog framework is designed to deploy across multiple systems, and it does this by using RMI as the transport between components instantiated on the different systems.

Unlike Ant, SmartFrog components do not offer public methods to set attributes. Instead, when the components wish to determine the value of attributes or enumerate components nested inside, they must ask the SmartFrog engine what those values are. The benefits of this are mixed. There is less automatic binding, but it does permit the task to be very dynamic in what it asks for. When one considers that a well-written Ant task needs to begin its `execute()` method with validation checks for its state, the overall amount of coding is about the same.

Why Ant Tasks for SmartFrog?

The SmartFrog Ant tasks enable developers to integrate SmartFrog-managed deployments with an Ant base build process. This lets one deploy to local or remote systems simply by invoking the appropriate target, perhaps even from inside the developer's IDE of choice.

Integrating deployment into the development cycle transforms functional testing. Imagine a build cycle in which an EAR file is created, deployed to a local host, then functional tests are run. SmartFrog can do that. Imagine taking that same build file, and having it deploy to a staging machine on site, and then to the live deployment on a server at a colocation site. SmartFrog can do that. Then imagine running JUnit tests against the staging machine, from a remote box in a different location, such as one hosted at an ISP. SmartFrog can do that, too. JUnit tests are simply different programs to deploy, after all.

Ant can be used for deployment today. Java Development with Ant devotes two whole chapters to the topic. So sometimes I get asked, "Steve, why don't we just use Ant, as that book tells us". To which I reply, "I wrote those chapters, and in the next edition of the book, I will replace the second chapter with an introduction to SmartFrog instead". Ant works well for simple deployments -such as to a local Tomcat server. You copy the files somewhere, you send a <get> request to a URL, and voilà, your application is deployed.

Where Ant is weak is that it does little for long-haul deployment, except for letting you use SSH to talk to a terminal on the remote system. All you can do there is run shell scripts or Ant itself, with little control as to what is going on. And because there is very little workflow in core Ant -other than conditional targets and the `<fail>` task, there is little that can be done when things start to fail.

Where it is most weak is configuration. Ant is about executing the steps to build something, and copying and running code can be forced into its build process. What you can not do is configure what you are deploying. Because SmartFrog can synchronise the configuration values of many components, even across multiple hosts, it makes it possible to deploy different parts of a bigger system, without spending hours tracking down configuration errors.

Introducing the Ant Tasks

The Ant tasks for SmartFrog are a set of tasks to deploy and manage SmartFrog-hosted applications from inside Ant. This integrates SmartFrog into the build process, and makes it easy to do advanced deployment from within an IDE, or from the command line.

Setting up

Setting up the Classpath

The SmartFrog Ant tasks need the following JAR files on their classpath

- `smartfrog.jar` – the core SmartFrog code
- `smartfrog-tasks.jar` -the ant tasks

When starting a daemon or running an application inline, the classpath must also include any libraries that are needed by the application. This can be done by

1. declaring them on the `<taskdef>` classpath (which implies that they must already have been compiled at that point)
2. Declaring them with a `<classpath>` element in the ant tasks which do the work
3. Using SmartFrog's dynamic class loading feature.

During development, option (2) is the easiest, through option (3) is useful for dynamic code generation.

In these code samples, we assume that the property `${smartfrog.dist.lib.dir}` is already set the `smartfrog/dist/lib` directory, and the property `${smartfrog.tasks.jar}` bound to the location of the tasks jar:

```
    <path id="smartfrog.classpath">
      <fileset dir="${smartfrog.dist.lib.dir}">
        <include name="**/*.jar"/>
      </fileset>
    </path>

    <path id="smartfrog.tasks.classpath">
      <path refid="smartfrog.classpath"/>
      <pathelement location="${smartfrog.tasks.jar}"/>
    </path>
```

Declaring the Tasks

Having set up a path to point to the SmartFrog jars, declaring the tasks is easy.

```
    <typedef
      resource="org/smartfrog/tools/ant/tasks.properties"
      classpathref="smartfrog.tasks.classpath"
      />
```

You can also use the new Ant1.6 library feature, which

```
    <typedef uri="http://smartfrog.org/"
        resource="/org/smartfrog/tools/ant/antlib.xml"
        classpathref="smartfrog.classpath"
        xmlns:sf="http://smartfrog.org/" />
```

The current release of the Ant task suite does not declare anything other than the same tasks as the properties file, though this may change in future. One difference is that the property file declaration gives all tasks an `sf-` prefix, such as `sf-deploy`. The XML declaration uses the namespace used in the `<typedef>` call to provide a unique prefix. For example, with the `xmlns:sf` declaration above , the prefix is `sf:`, so the same task is called `sf:deploy`.

Remember that the scope of a namespace declaration does not cover the remainder of the document, but only that bit of the XML document that is nested inside the declaration. Declaring the namespace in the `<taskdef>` call is the wrong place -declaring it at the target or project level is the appropriate action.

Finally, note that you can also use the 'antlib' autoloading feature of ant, documented in the Ant manual [Apache03b]. Here the act of declaring the package name of the tasks, in the `antlib:` schema, is sufficient for them to be loaded and used.

```

<project default="deletetest" xmlns:sf="antlib:org.smartfrog.tools.ant">
```

For this to work, the relevant JAR files must be on Ant's classpath. Rather than copy them in to the `ANT_HOME/lib` directory, we recommend that you use the -lib option to list a directory containing the SmartFrog-specific JAR files.

We tend not to use this ourselves much as (a) we are sometimes compiling the tasks as dependent libraries inline, in code that shares our common taskdef targets, and (b) it does not work so well with IDEs, where the `-lib` options cannot be set

Running an Application

When you "run" an application, the task creates a standalone SmartFrog daemon that runs an application until it terminates (or is terminated). That means the task is blocked.

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

This technique is sometimes useful, as it removes the need to run a daemon first. However, because the task blocks until completion, one needs some complex `<parallel>` logic in the build file to work with the deployed program.

Note that in this example we have turned assertions on. This is for debugging -it enables Java1.4 assertions. The syntax of this nested type is documented in Ant's `<java>` task, and, for the curious, is also found in `<junit>`.

Deploying an Application

Deployment is of course the core binding of SmartFrog to Ant. The `<sf-deploy>` task does this. It deploys an application to a running SmartFrog daemon -on any host you care to name. This is the key -you can deploy to a remote box at the far end of a network *almost* as easily as deploying to a local system. The sole difference is that you have to make the files available to the far end. Let's start with local deployment where the relevant SmartFrog component classes are already installed in the running machine, which is the case if they are in the classpath of the daemon on startup.

One important thing to remember is that the `<sf-deploy>` task starts a new SmartFrog engine, solely for the deployment. It is this process -with its own classpath- which expands the deployment descriptor, resolving all references, producing a descriptor which is then sent to the destination host, either local or remote.

Deploying a resource

When you deploy a resource, the SmartFrog descriptor must be be in a file on the classpath of the sf-deploy task. You can add a new classpath inline, if you need to, or a reference.

```
      <sf-deploy logStackTraces="true" classpathref="my.new.classpath" >
        <application name="app" descriptor="/org/example/router.sf"/>
      </sf-deploy>
```

To deploy to a remote system, set the host attribute. Here we deploy two applications to a machine called "gecko":-

```
      <sf-deploy host="gecko" >
        <application name="router" descriptor="/org/example/router.sf"/>
        <application name="server" descriptor="/org/example/server.sf"/>
      </sf-deploy>
```

Even when deploying to a remote system, resources are resolved locally, in the <sf-deploy> process, as are `#include` references in the descriptors themselves.

Deploying Files

It is as easy to deploy a descriptor in a file as it is a resource:

```
      <sf-deploy host="gecko">
        <application name="app" file="application.sf"/>
      </sf-deploy>
```

Inline Descriptors

The Ant tasks allow you to insert descriptors inline inside the build file. This gives you the power to insert any text into the descriptor from Ant properties. It is ideal for taking a base descriptor file, including

```
<application name="${application.name}">
       #include "org/example/service.sf"
       AxisHome ${axis.home};
```

```
</application>
```

Although this is occasionally useful, it is somewhat inelegant. The flaw with such inline descriptor creation is that it makes ant the *only* way to deploy the application. While this is useful during development, in production systems the command line and GUI are also useful, as is the default file for a daemon to load on startup.

Furthermore, there is a structured way of passing configuration properties to a deployment descriptor which works in all these deployment mechanisms.

Because of this, we recommend that you use the inline technique primarily for generating test-time deployments -not staging or production ones- or if you want to do more advanced dynamic code generation.

Passing Properties

The SmartFrog languages has the `PROPERTY` and `INTPROPERTY` keywords, which bind attributes to property, as strings and integers, respectively. You can pass properties to a deployment descriptor using the `<sysproperty>` element:-

```
    <sf-deploy classpathref="run.classpath"
       host="${deploy.host}"
       >
       <sysproperty key="axis.home" file="${lib.dir}" />
       <sysproperty key="users" value="5" />
       <application name="${application.name}">
         #include "org/smartfrog/services/cddlm/cddlm_on_jetty.sf"
         AxisHome PROPERTY axis.home;
        users INTPROPERTY users
       </application>
    </sf-deploy>
```

The element is documented in the Ant manual; it is exactly the same as that supported by the `<java>` task.

The `<sysproperty>` element is supported by `<sf-deploy>`, `<sf-undeploy>`, `<sf-run>` and `<sf-startdaemon>`. The properties it sets are normal Java system properties. This allows the element to be used for configuring any aspect of SmartFrog which is configurable by such properties. Many of the attributes of the <sf-deploy> and <sf-daemon> tasks are mapped directly into Java properties, for this reason.

There is an extra feature in SmartFrog configuration which is not (yet) in the `<java>` task. The `<propertyfile>` element enables the task user to pass a file of Java properties straight to the program. It has two attributes, `file` and `optional`. The former names the file to use, while the optional flag can be used to indicate that it is not an error for the file not to exist.

We often use this when starting a daemon, to provide a point for per-user, per-project customisation:

```
        <propertyfile file="${runtime.properties}" optional="true"/>
```

Remember that Property files are ASCII, with escaping needed for special values in the property names or values. The rules are specified in the `java.util.Properties` javadocs.

Time-limited deployment

```
      <sf-deploy timeout="30">
        <application name="app" file="broken-file.sf"/>
      </sf-deploy>
```

Timeout is measured in seconds.

Specifying a timeout during deployment does not control the lifetime of an application. Instead it tells ant that if the spawned deployment process takes longer than a certain time -here thirty seconds- it should be terminated, and task considered to have failed.

Setting timeouts is very brittle -slow machines or networks can cause unexpected timeouts. At the same time, it is useful to have the option for fully automated builds. The best solution may be to have a very long time out, such as "3600" -an hour – just to cope with emergencies.

The timeout attribute is available on all the main tasks, incidentally. It is most useful on `<sf-run>`.

Waiting for an application

There is no explicit support for this.

We recommend that you use `<waitfor>` with a condition that indicates availability of the system being deployed. For example, a listening socket, or a downloadable web page.

Undeploying/Terminating an Application

A application is terminated by undeploying it, using the `<sf-undeploy>` task.

```
      <sf-undeploy host="gecko" application="test"/>
```

To avoid an error if the application was not deployed, set the `failonerror` attribute to false.

```
      <sf-undeploy host="gecko" application="test" failonerror="false"/>
```

Starting a SmartFrog daemon

```
      <sf-startdaemon classpathref="run.classpath"
        logStackTraces="true" spawn="true">
          <!-- assertions are enabled -->
        <assertions enableSystemAssertions="true">
          <enable/>
        </assertions>
          <!-- load in a property file if it is present -->
        <propertyfile file="${runtime.properties}" optional="true"/>
      </sf-startdaemon>
```

Stopping a SmartFrog daemon

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

Setting the `failonerror` flag means that this target does not break the build if called and there is no daemon already running.

Note that one can also undeploy an application by undeploying the application called `"rootProcess"` :-

```
<sf-undeploy application="rootProcess" timeout="${medium.timeout}"/>
```

The `timeout` attribute sets a limit for the undeployment to take before Ant build file kills the spawned undeployment process.

Probing for a Daemon

```
    <presetdef name="sf-daemonfound">
      <condition >
        <socket port="3800" server="localhost" />
      </condition>
    </presetdef>
```

An example of its use is the following pair of targets. The first probes for the daemon existing, and the second starts it if one was not found.

```
  <!-- =========================================================== -->
  <!--  look for a local daemon. Sets the property  local.daemon.running if
    one is listening on port 3800-->
  <!-- =========================================================== -->
  <target name="probe-local-daemon" depends="init">
    <sf-daemonfound property="local.daemon.running" />
  </target>

  <!-- =========================================================== -->
```

```
<!-- conditionally start the daemon if one was not found already-->
<!-- ========================================================== -->
<target name="start-daemon-if-needed"
  depends="declare-extended-smartfrog-tasks,probe-local-daemon"
  unless="local.daemon.running">
    <sf-startdaemon />
</target>
```

Waiting for a Daemon

There is no explicit task to probe for a server. Here is an ant1.6 preset to wait for ten seconds for one to appear at the local host, so as to synchronise a daemon startup with the next stage of a deployment:

```
  <!-- wait for 10 seconds for a daemon. Set maxwait to a different
  value for more or less time, timeoutproperty to the name of a property
  to set on failure -->
  <presetdef name="sf-waitfordaemon">
      <waitfor maxwait="10" maxwaitunit="second">
          <socket server="localhost" port="3800"/>
      </waitfor>
  </presetdef>
```

Validating SmartFrog Files

```
      <sf-parse quiet="true">
          <classpath path="${target.jar}"/>
          <source dir="${build.classes.dir}"
              includes="org/cddlm/components/*.sf">
          </source>
      </sf-parse>
```

Advanced Techniques

Spawned SmartFrog daemons

The `<sf-daemon>` task supports a `spawn` option, which decouples the daemon from the Ant runtime. Even after Ant finishes, the daemon continues to run.

Here are some warnings about this option

- It works best with Ant1.7 or later
- All console output from the daemon is lost.

Debugging SmartFrog-JVM code

To debug code hosted inside the SmartFrog JVM, the following process works well.

1. Run SmartFrog as a daemon, under the debugger.
2. Run the Ant target to perform the deployment

or

3. Include the JAR files in the daemon, and add the deployment command string to the debugger command line.

Option (1) and (3) work best if the files exist when the run begins, as the debugger can be pointed at the source files. Option (2) gives one the ability to debug the deployment process, and to hot-update code without halting the debug session.

Contributing to the SmartFrog Tasks

Being an open source project, SmartFrog depends on contributions from the user community. Feedback and bug reports are the simplest contributions, but we welcome enhancements to the code and documentation of the ant tasks.

To get involved, raise the subject on the SmartFrog developer list. You will need read-only CVS access to the SmartFrog source repository on SourceForge, or a source distribution.

All ongoing task development is done against the CVS_HEAD version of Ant, as this enables us to feed changes into the Ant repository to improve the SmartFrog development and test process, including the tasks. Yet the tasks must also support the current shipping version of Ant: we do not want to require the use of pre-release build tools in our own product.

Security

This release of the tasks has security support built in. A `<security>` element can define the keystore, principal and password to use when signing. This element is actually a datatype and can be declared outside a deployment task, then reused via `<security refid>` when needed.

attribute	***meaning***
`securityFile`	File containing security properties
`keyStore`	keystore file for signing
`alias`	Alias to use when signing
`policyFile`	Optional file of security policies, passed to the JVM as `-Djava.security.policy==${policy.file}`

The `securityFile` attribute points to a property file that contains security data, especially

```
org.smartfrog.sfcore.security.keyStorePassword=MkgzZVm9tyPdn77aWR54
org.smartfrog.sfcore.security.activate=true
```

This is passed to the JVM, avoiding placing the values on the command line (for security reasons).

The policy file is not interpreted by SmartFrog; it is for the JVM itself.

Not all tasks need all the security settings. For signing, only the `keystore`, `alias` and `securityFile` is needed (the latter for password). When starting a daemon securely, or deploying to it, the `alias` and `policyFile` attributes are optional. If a `policyFile` is set it is applied; the alias is ignored completely.

One feature of the datatype is that all these values can be omitted to produce an empty security.

```
<security id="default.security" />
```

This permits other tasks (such as signing) to use the security datatype without caring whether it is empty or not. Empty security declarations are treated as *security off*.

All the deployment tasks take a security element. The `<sf-sign>` task requires one, for signing JAR files. However, if this task is passed an empty security datatype, it does not sign it, instead printing a message notifying that signing was skipped. To continue our example, this signs the JAR files in the distribution directory, if the default security attribute is set.

```
<sf-sign >
 <fileset dir="dist/lib" includes="**/*.jar" />
 <security refid="default.security" />
</sf-sign>
```

We may tune this no-op behaviour and other aspects of the type based on user feedback. For example, the password could be supported as an attribute; the security file being dynamically generated. We may also add an `enabled` flag that turns security on or off at the flick of a switch. Feedback is encouraged.

Warning:

Because Java does not have good control over Unix (or WinNT) file permissions, Ant cannot create files with explicitly controlled permissions. A `<chmod>` operation can restrict access, but only after creation. The only way to securely create property files is to create them in a directory in which only the owner has access. Such a directory can be created with Ant on Unix, but not on Windows NT-based platforms.

Futures

These are some things that we hope to add in the next release. If you have a pressing need for them, please check with the developer mail list to see if they have been implemented, and if not, please speed up the process by authoring and contributing them to the project (with tests please!)

Probing for a deployed application

There is no way to ask for a named application other than undeploying it, or attempting to deploy one of the same name. We would like something less complex, with fewer potential side effects.

Retrieving information from an application

After probing for an application comes the act of retrieving information from a local or remote daemon, such as the latest value of a reference.

Reference

1. [Apache03a] *Apache Ant* http://ant.apache.org/manual/
2. [Apache03b] *Antlib* http://ant.apache.org/manual/CoreTypes/antlib.html
3. [HL02] Hatcher E., and Loughran S., *Java Development with Ant*,