Java Execution Tasks

These tasks make it easier to execute Java programs in separate processes.

Java Component

The Java component runs Java Programs. It is an extension of the `runshell` component, so all attributes associated with that are usable. the exception is the `shellCmd` attribute, which is rebound to the JVM

Attributes

Attribute	*function*	*required?*
`classname`	name of the class to execute	one of this or jar
`jar`	location of a JAR file to run.	one of this or `classname`
`endorsedDirs`	list of endorsed directories. These are the only directories that can contain "endorsed JAR files", as defined in Java1.4 and later. Any files that are passed in here are recognised as such, and the directory that contains them marked as endorsed. Duplicate directories are also recognised.	
`classpath`	list of JAR files and directories	
`sysproperties`	list of properties [-D] options as [name,value] tuples `[["http.proxy","true"],["http.proxyPort",8080]]`	no
`environment`	list of environment variables [-D] options as [name,value] tuples `[["PATH","/lib/path;~/bin"]`	
`jvmargs`	list of arguments to the JVM `["-incGC","-server"]`	
`jvm`	name of the program to start the JVM. Default is `"java"`.	
`assertions`	assertions on/off flag	no, default=false
`maxMemory`	maximum memory (in MB). Remember to allocate more for 64-bit systems, as the memory increases.	

Implementation

The Java component uses the normal `RunShell` component to perform its work, but builds up a new command line combing the Java command, the arguments to the JVM, and the arguments to the programs themselves. Classpath setup is another feature of the component.

Java Package Component

This is a helper component, to aid with classpath setup.

Every Java package represents a set of one or more JAR files.

LoadClass Component

The component loads a Java class, and optionally, tries to create an instance using an empty constructor. It

forces these classes into the system, so can be used to ensure that classes are available for other programs.

It serves the following purposes

1. Verifies that classes can be loaded; that the classpath is correct and all needed libraries are present.
2. Force loads classes into the classpath
3. Verifies that a class can be instantiated
4. Forces any static construction logic of a class to take place, so creating singletons and running any static or shared initialisation logic.

Classes will be retained for the duration of the lifecycle

```
LoadClassSchema extends Schema {
    classes extends Vector;
    create extends Boolean;
    retain extends Boolean;
}

LoadClass extends Prim {
    schema extends LoadClassSchema;
    sfClass "org.smartfrog.services.os.java.LoadClassImpl";
    create false;
    retain true;
}
```

Attributes

Attribute	*function*	*required?*
`classname`	List of classes to load `["java.lang.String"]`	Yes; can be empty []
`create`	Flag to indicate whether instances should be created or not.	Yes, default=false
`retain`	Flag to indicate whether classes and instances should be retained until the class is terminated.	

Implementation

All properties are retrieved in the `sfStart()` operation; then each class is loaded using the codebase of the component. After all classes have been loaded, if the create flag is set, then any instances are created.

Unless `retain` is false, the classes and instances are kept until the component is terminated, otherwise they are discarded immediately. A `System.gc()` call is made after discarding all references, but this is of course a hint -there is no deterministic way of purging all references inside a JVM.

Library Components

These are currently unsupported, experimental components to retrieve JAR files from the Maven1/Maven2 libraries, such as the ibiblio.org library: http://ibiblio.org/maven2.

As they are unsupported, users are invited to examine the components in libraries.sf, and the implementation classes and examples to determine their use. As they stabilise and get used more, they will become documented.

```
#include "/org/smartfrog/services/filesystem/components.sf"
#include "/org/smartfrog/services/os/java/components.sf"
#include "/org/smartfrog/services/os/java/library.sf"

sfConfig extends Compound {

    sfSyncTerminate true;

    library extends Maven2Library {
    }

    commons-logging extends JarArtifact {
        library LAZY PARENT:library;
        project "commons-logging";
        version "1.0.4";
        sha1 "f029a2aefe2b3e1517573c580f948caac31b1056";
```

```
        md5 "8a507817b28077e0478add944c64586a";
    }

    axis extends JarArtifact {
        library LAZY PARENT:library;
        project "axis";
        artifact "axis";
        version "1.1";
        sha1 "edd84c96eac48d4167bca4f45e7d36dcf36cf871";
    }


    tcpmonitor extends Java {
        classname "org.apache.axis.utils.tcpmon";
        classpath [
            LAZY axis:absolutePath,
            LAZY commons-logging:absolutePath];
    }

}
```

There are some issues to be aware of when using these components:

- Security in the ibiblio.org site is known to be inadequate; too many people have access to that site.
- If the MD5 or (better), sha1 checksum of the component is declared, then these are verified in the descriptor. This does secure the downloads.
- Most JAR files in the central MD5 repository are unsigned; none will be loaded by SmartFrog's own classloader when running securely.
- It can take a while to download files over HTTP. Once they have been downloaded into the local cache (`${user.home}/.m2/repository`), they will be available for all applications, without more downloads.
- The real Apache Maven2 tools treat files with a version -SNAPSHOT specially, with intermittent polls for updates. The SmartFrog components do not. Trying to deploy -SNAPSHOT releases is considers dangerous as they can vary from day to day, making deployment inherently unstable.

The two ways to use this component effectively are:

1. Have a private repository of JAR files, in which each one is signed by an authority trusted by the SmartFrog runtimes. This is the only way to load classes into a secure classloader.
2. Use the components to set up the classpath for a `Java` component. This is what is done in the example above.