1 Declarative Package Loading

1.1.1 Steve Loughran
2004-09-29

This document discusses better ways of packaging JAR files for classloading in SmartFrog

1.1.2 The problem

Add-on components often re-use common libraries. For example, Xerces and Xalan are ubiquitous, Ant can be used by Jetty, Tomcat and CruiseControl. To date, our components always include these libraries in their lib dir, under SCM, and rely on them being copied into the runtime at deploy time, or for the appropriate `sfCodebase` URL to be generated during the build process.

What would be better would be if:

1. Every component could declare that they depend upon libraries supplied by other components
2. The classpath of dependent components would be set up to refer to the locations of the packages they depend on.
3. It would be nice if something could actually demand-locate JAR files in a network repository
4. We could include version checking, to the best of our ability.

1.1.3 First Solution: Package Components

A `JavaPackage` component is a simple extension of Prim; it does

```
JavaPackage extends Prim {

    sfClass "org.smartfrog.services.os.java.JavaPackageImpl";

    javaPackageSchema extends Schema {
        //list of files, packages or URLs
        source extends OptionalVector;
        //a string classpath that is generated automatically
        //classpath extends OptionalString;
        //lazy attribute
        //classpathList extends OptionalVector;
        //classes to look for
        requiredClasses extends OptionalVector;
        requiredResources extends OptionalVector;
        useCodebase extends Boolean;
    }

    //we don't use the codebase
    useCodebase false;

}
```

The health check for the component would verify that all classes named in the `source` vector existed. We could use `sfCodebase`, but that attribute must also contain the `JavaPackage` implementation.

This component would be used in others

```
XercesPackage extends JavaPackage {
  source "xerces.jar";
  requiredClasses ["org.apache.xerces.XercesImpl"];
//classpath: sfDeploy() instantiated attribute listing classes
}

AxisPackage extends JavaPackage {
  source ["axis.jar",XercesPackage];
  requiredClasses ["org.apache.axis.AxisEngine"];
```

```
}

AxisEndpoint extends Prim {
 sfCodebase [AxisPackage];
 //...
}
```

The idea is that by referring to your dependent components, you could get your classpath set up right. But what if you want to use the default classpath? You may still have dependencies on code that must be there, and therefore still benefit from explicit declaration of dependence. Similarly, if we add version information to the declarations, we could use this to probe JAR files and verify their age.

This would need, at the very least, the classloader to be enhanced to handle a list as an alternative to a string. The handling logic of the list would be

1. `String` elements are treated as containing one or more URLs to content.
2. Elements that implement the `File` interface have their `absolutePath` attribute converted to a URL to a JAR file. Or we extend the interface to have a URL attribute and use that, which could be more flexible.
3. Elements that implement the Package interface, have their `classpath` attribute read and the contents inserted into the list
4. Elements that are in fact lists themselves are recursively parsed.
5. Any other type of entry is an error.
6. All duplicate entries are eliminated; duplication is based on case-sensitive string comparison of URLs

This would be enough to resolve paths, provided that package components had their classpath attribute set when needed. The default definition would mark it as lazy, with the appropriate consequences. This is to enable future extensions to actually set the classpath attribute with values other than the value of `sfCodebase`, and do so during deployment.

This initial solution would do nothing to ensure that needed files were present on the machine, or up to date.

1.1.4 Versioning

Could this solution support versioning?

It already supports it to the extent that one could explicitly bind a component to a package representing a particular file. If different packages with different files and names were declared -such as Xerces1, Xerces2, then components could declare their dependence upon the relevant package.

While this makes versioning explicit, it has the side effect of binding version options with the rest of the deployment/configuration data, which may or may not be what is desired. Given that one also tends to have a binding at build time, there is now duplication of information.

More important than the duplication, the configuration data may be scattered in many places; it may seem better to have a mapping file that explicitly mapped names of libraries used in a system to the versions that were needed. Of course, there is the extra challenge that a sufficiently large system will want to use different versions of libraries in different places, so the mapping file may need overriding. Perhaps the solution is just to integrate the version dependencies with the rest of the configuration data.

Let's assume that components include version dependencies in the declarations of which package they depend upon. To be precise, each package component includes an optional `minVersion` attribute that declares the minimum version of a package which is to be supported, and a very optional `maxVersion` attribute to declare the maximum that is allowed.

1.1.5 The Repository

If package declarations declare version information, then we can resolve the URLs of needed packages not

through explicit path strings in the package, but via a repository component. This component would

1. Offer a mechanism for packages to locate a library that they need in a list of one or more local directories.
2. Use introspection into JAR manifests to determine version information
3. Support live update of the directories; polling or a `touch()` operation would trigger a reload.

With a repository, components (including the Java component) would declare their dependencies on packages, without the packages including a declaration of the location of their files. Instead, the repository would locate the files from the set of files it knew of.

A repository may also like to be able to delegate queries to one or more relay repositories.

We could integrate the repository with the SmartFrog command line, with a new parameter, `-repository <dir>` which would take a path to a directory. Every occurrence on the command line would create a new repository, perhaps listed on the `sfRepository` chain.

Repository support does seem to complicate the deployment process, as the repositories must be up and running before dependent components can be instantiated.

1.1.6 Remote Repository

A Remote Repository would be a component that actually retrieved JAR files from a remote repository, placed them somewhere visible, and returned the URLs to the files. This could be as simple as doing a lookup to determine the URL of the files on a public server, or it could actually download the files to the local file system.

URL lookup is the simplest; we could have an HTTP daemon that served up files, and listened to a request to look for a particular file, returning the URL of the aforementioned file, or an error code. We'd use Anubis as the means of locating daemons to query; it would also support an RMI interface for SmartFrog to use for the lookup.

A fetching repository would actually pull files from a remote system to a local cache; some simple timestamps/etag mechanism could be used to keep the cache up to date.

The Apache Maven repository system uses the 4-tuple of (project, artifact, version,extension) to determine what to download. A URL is built up of http://repository/project/artifact-version.extension and then timestamp-downloaded to the local filesystem.

1.1.7 Security

There should be no security implications, as when security is turned on, all JAR files have their signatures verified. It may be wise to disable remote repositories unless security is enabled, however.

1.1.8 Integration with the Java component

While packaging and repository support would be useful during component deployment, where it would be very interesting would be when executing other Java programs, such as through the Java component.

This component will not need the packages until started, but it will then need all packages as local files, not as remote URLs, because it will need to pass them on the classpath of the new JVM.

This implies that we will need a way to get files from URLs. If the files a local file:// URLs, this is trivial; for remote files we will need to fetch them and store them locally. This argues in favour of a package retrieval component that retrieves all remote URLs in its classpath list, and creates a new classpath consisting entirely of local ones.