Building SmartFrog

27 Jul 2007

1 Introduction

This document describes how to build the SmartFrog framework from its source files. It describes how to set up the development system, and how to execute the build scripts.

In theory, SmartFrog can be built on any platform capable of running a current version of Java. In practice, we develop and test on Linux, Windows, and Mac OS/X. We recommend a system capable of running Java at a reasonable speed. We also advise a recent Java runtime and a good IDE.

SmartFrog has been developed using Emacs, jEdit, Intellij IDEA, Eclipse and NetBeans. The exact details of how to set up IDEs for SmartFrog is covered in a separate “IDE development” document.

2 Tools

2.1.1 Java JDK

The minimum platform for building SmartFrog is the Java JDK version 1.4.2 is ; Java 5 or Java 6 is preferred. The full JDK must be installed, not just the JRE.

Windows:

Try not to install Ant in a directory with spaces in it, as things break sometimes.

On Windows, after installing the JRE, consider removing the update program that is set to run on startup. If you are developing Java code, you do not want a remote team to be forcing JRE updates onto your system, except under your own control. The same applies for server installations, of course.

Linux

On Linux, we have found the JPackage distributions to be the most reliable installations. Do not attempt to install the Sun RPMs onto an Ubuntu desktop using `alien`; use the Debian packages which are provided, either on the main servers or (for older installations), on the backports server.

Alternate Java runtimes, such as Kaffe and Apache Harmony are not explicitly supported, especially at build time. Any issues with running SmartFrog under these JREs should be filed as defects, but developers must be aware that the Sun distribution is the sole recommended Java implementation to use at build build time.

On a system with multiple JREs installed, it is imperative that the Sun JDK gets priority. JPackage-installed installations of gcj can interfere with this, so please uninstall gcj when trying to get the the Sun JRE to work, or edit the bindings in `/etc/alternatives`.

2.1.2 JDK Configuration

Once the JDK is installed, set the `JAVA_HOME` environment variable to point to the installation directory, which, in Linux, can be done with

```
export JAVA_HOME=/usr/java/jdk1.6.0
```

Also set the PATH environment variable to include the Java binaries directory, e.g.

```
export PATH=$PATH:$JAVA_HOME/bin
```

(For Windows, just use the method of setting environment variables supported by your version of the OS. On Windows XP, this is accessed via “My Computer” / right clicks for "Properties" / "Advanced" tab / "Environment Variables" button.)

To test that the JDK is correctly installed, type `javac -version` at the command line. The output should

resemble the following

```
> javac -version
javac 1.6.0
```

Similarly, the `java` command itself must be on the command line

```
> java -version
java version "1.6.0"
Java(TM) SE Runtime Environment (build 1.6.0-b105)
Java HotSpot(TM) Server VM (build 1.6.0-b105, mixed mode)
```

If a different JVM is shown from that expected, it is likely that there is more than one JVM installed on your system, and the path is picking up the wrong version.

2.1.3 The CLASSPATH environment variable

Please avoid adding files to the CLASSPATH environment variable. IT only causes confusion, and stops tools such as Ant and SmartFrog from managing the classpath. Be warned that third party products (such as Apple iTunes/QuickTime) can create this variable. It should not exist:

```
>echo $CLASSPATH

>
```

Or, in windows

```
>echo %CLASSPATH%
%CLASSPATH%

>
```

Once the JDK is working, move on to setting up Ant

2.2 Ant

We require Apache Ant version 1.7.0 (or later), from http://ant.apache.org. Try not to install Ant in a directory with spaces in it, as things break sometimes.

Once installed,

- Set the `ANT_HOME` environment variable

  ```
  export ANT_HOME=/usr/bin/apache-ant-1.7.0
  ```

- Add the Ant binary directory to the PATH, e.g.:

  ```
  export PATH=$PATH:$ANT_HOME/bin
  ```

This can be tested by typing ant -version on the command line

```
> ant -version
Apache Ant version 1.7.0 compiled on December 13 2006
```

An out of date version can often be caused by a script installed in a .deb or RPM package taking priority over any local installation.

Running ant -diagnostics can identify many potential issues with building SmartFrog. To work, the build requires

1. Network access to the Java artifact repository on ibiblio.org
2. Required JAR files for certain Ant tasks.

The needed JAR files can be retrieved from remote sites, but only once any proxy settings have been passed to Ant.

2.2.1 Ant Proxy Setup

If you develop behind a firewall that needs a proxy to access the outside network, then Ant must be configured with the local proxy configuration. Although Ant1.7 beta attempted to use Java 5's automatic proxy setup, this turned out to cause more trouble than expected, so has been turned off. To use Ant and Ivy behind a proxy-bridged network, the proxy must be set by hand by editing the ANT_OPTS environment variable

In Linux, this is done with another environment variable definition in ~/.bash_profile:

```
export ANT_OPTS="-Dhttp.proxyHost=web-proxy -Dhttp.proxyPort=8088"
```

In Windows, this can be set in the environment variables dialog box. Do not quote the proxy string in this situation. The complete list of proxy properties is documented by Sun: http://java.sun.com/javase/6/docs/technotes/guides/net/proxies.html.

To check that the proxy is configured, run `ant -diagnostics`, and look at the proxy settings:

```
------------------------------------------
 Proxy information
------------------------------------------
http.proxyHost = "web-proxy"
http.proxyPort = "8088"
Java1.5+ proxy settings:
HTTP @ web-proxy:8088 web-proxy:8088 [unresolved]
```

This says that the http proxy settings have been picked up, and propagated to the Java1.5 proxy libraries.

If these properties were undefined, and the Java1.5+ settings stated that the JVM had a direct connection, then proxy information has not been picked up. Unless the build is taking place on a machine with unrestricted network access, the build will fail

2.2.2 JUnit

Ant uses JUnit to run its tests. Before Ant1.7, JUnit had to be installed in ANT_HOME/lib; Ant 1.7 uses reflection to pick up the JAR if it is on the classpath of the executed tests. However, we cannot guarantee that everything works in this set up; we have not tested it enough. It is safest to download junit3.8.2 and add it to Ant.

Please do not use JUnit 4.x with SmartFrog; it is not supported.

2.2.3 Ivy

SmartFrog currently uses Ivy 1.4.1, the last pre-Apache release. As soon as a full Apache release ships, we will migrate. For now, there is a copy of Ivy in core/antbuild/ivy, which components explicitly load. If there is a version of Ivy on the classpath, it may be picked up instead of this version.

2.3 Open SSL (Unix systems only)

OpenSSL is only required if you need to create a certification authority for securing SmartFrog hosts and SmartFrog code. Most of the build scripts will run without it, so you may not need this to get started. OpenSSL (version 0.9.5a or later) is required to set up a certification authority to initialize the security infrastructure.

The OpenSSL binaries directory must be added to the system `PATH`. This is automatic if the package is installed using apt-get or an RPM manager. Test for it on the command line

```
> openssl version
OpenSSL 0.9.8a 11 Oct 2005
```

Open SSL is not needed to sign artifacts, once a Certification Authority has been created on a Unix system.

3 Subversion Access

The complete SmartFrog source can be checked out from the SourceForge Subversion repository by following

the instructions given below.

```
svn checkout https://smartfrog.svn.sourceforge.net/svnroot/smartfrog/trunk/core core
```

To update your working copy:

```
svn update
```

Developers with commit access to the repository can commit their changes through

```
svn commit -m "insert your message here"
```

To browse the repository, browse to http://smartfrog.svn.sourceforge.net/viewvc/smartfrog/trunk/core/

To get more information about Subversion visit: http://svnbook.red-bean.com/

Subversion Line ending setup

In `~/.subversion/config`, you must set up the end of line policy for .sf files

```
[auto-props]

*.sf = svn:eol-style=native
*.pl = svn:eol-style=native
```

You may also want to add the recommended Apache line ending rules

```
INSTALL = svn:eol-style=native
KEYS = svn:eol-style=native
Makefile = svn:eol-style=native
README = svn:eol-style=native
*.aart = svn:eol-style=native
*.ac = svn:eol-style=native
*.am = svn:eol-style=native
*.bat = svn:eol-style=native
*.c = svn:eol-style=native
*.cat = svn:eol-style=native
*.cgi = svn:eol-style=native
*.classpath = svn:eol-style=native
*.cmd = svn:eol-style=native
*.cpp = svn:eol-style=native
*.css = svn:eol-style=native
*.cwiki = svn:eol-style=native
*.data = svn:eol-style=native
*.dcl = svn:eol-style=native
*.doc = svn:mime-type=application/msword
*.dsp = svn:eol-style=CRLF
*.dsw = svn:eol-style=CRLF
*.dtd = svn:eol-style=native
*.egrm = svn:eol-style=native
*.ent = svn:eol-style=native
*.ft = svn:eol-style=native
*.fn = svn:eol-style=native
*.fv = svn:eol-style=native
*.grm = svn:eol-style=native
*.g = svn:eol-style=native
*.h = svn:eol-style=native
.htaccess = svn:eol-style=native
*.html = svn:eol-style=native
*.ico = svn:mime-type=image/x-icon
*.ihtml = svn:eol-style=native
*.in = svn:eol-style=native
*.java = svn:eol-style=native
*.jpg = svn:mime-type=image/jpeg
*.jsp = svn:eol-style=native
*.js = svn:eol-style=native
*.junit = svn:eol-style=native
*.m4 = svn:eol-style=native
*.mf = svn:eol-style=native
*.MF = svn:eol-style=native
*.mod = svn:eol-style=native
*.pdf = svn:mime-type=application/pdf
*.pen = svn:eol-style=native
*.pl = svn:eol-style=native
```

```
*.png = svn:mime-type=image/png
*.pod = svn:eol-style=native
*.project = svn:eol-style=native
*.properties = svn:eol-style=native
*.py = svn:eol-style=native
*.rb = svn:eol-style=native
*.rdf = svn:eol-style=native
*.rnc = svn:eol-style=native
*.rng = svn:eol-style=native
*.rnx = svn:eol-style=native
*.rss = svn:eol-style=native
*.sh = svn:eol-style=native
*.svg = svn:eol-style=native
*.tld = svn:eol-style=native
*.txt = svn:eol-style=native
*.types = svn:eol-style=native
*.xcat = svn:eol-style=native
*.xconf = svn:eol-style=native
*.xegrm = svn:eol-style=native
*.xgrm = svn:eol-style=native
*.xlex = svn:eol-style=native
*.xmap = svn:eol-style=native
*.xml = svn:eol-style=native
*.xsd = svn:eol-style=native
*.xsl = svn:eol-style=native
*.xslt = svn:eol-style=native
*.xsp = svn:eol-style=native
*.xul = svn:eol-style=native
```

These settings let subversion convert line endings to the native from across systems.

Subversion proxy setup

in `~/.subversion/server` declare different groups for each set of servers you need to set up proxies for:

```
[groups]
hp=*.hp.com
sourceforge=*.sourceforge.net
```

For each group, provide the specific proxy options:

```
[hp]

[sourceforge]
http-proxy-host=web-proxy
http-proxy-port=8088
```

This bypasses the proxy for internal HP sites, yet uses it for SourceForge.

4 A guide to the directories

The root directory is called `core` . This contains some useful files

file	*description*
`common.xml`	The build file <imported> by most projects, containing common targets.
`build.xml`	A build file which can be used to build or clean the entire project
`build.properties` (only present if manually created)	custom properties that are read for all projects

More important, It also contains a number of subdirectories

directory	*description*
`smartfrog`	The project that creates the core smartfrog JAR files
`components`	Sub projects for components; each component module has their own subdirectory under here. `libraries.properties` lists the versions of all JAR files that Ivy should retrieve on demand.
`testharness`	Test harness and unit/functional tests for SmartFrog.
`extras`	Parts of the SmartFrog framework that are not components. This includes IDE and build tool integration
`antbuild`	Extra files needed for building. antbuild/antlib : ant libraries antbuild/doc: documentation antbuild/ivy: ivy support antbuild/repository: artifacts that are not in the public repository antbuild/xml: more build files
`forrestsite`	Site definitions for Apache Forrest
`release`	A build file to create release artifacts
`releases`	The directory into which core/smartfrog creates releases
`external` (optional)	Build files under here with a corresponding ivy.xml file get included in the `published` target of `core/build.xml`. As a result, you can include external projects into the main build.

The normal structure of an individual project is as follows:

file/directory	*description*
`build.properties` (only if manually created)	Properties file overriding build file definitions
`build.properties` (optional)	Example build.properties file
`build.xml`	Main build.xml file
`buildRelease.xml`	Redistributable build.xml (if different)
`releaseBuild.xml`	Build file for creating a release. Found on some projects
`project-template.pom`	Projects have this file to provide a template of what to publish as metadata in a maven2-compatible repository.

During a build, other files and directories are created:

file/directory	*description*
`build`	Interim files created during a build. build/classes: classes build/test/classes: test classes build/test/data: XML test results build/test/reports
`dist`	Distributable artifacts. dist/lib: JAR files

file/directory	*description*
`version.properties`	Properties file containing the current SmartFrog version

These directories can be deleted when not needed; the `clean` targets will do this

5 Building

Building core/smartfrog

The first step to building is to create the standalone smartfrog.jar, sfServices.jar and sfExamples.jar files.

```
>cd core/smartfrog
>ant dist
```

This will build everything. If Ivy is installed (or reachable via a relative path), then the artifacts will be published to the local Ivy repository. Follow-on builds only work if the artifacts were published

Building the Ant tasks

If the core builds, try the Ant tasks

```
>cd core/smartfrog
>ant all
```

This is a follow-on, which tests that the ant tasks will build. This will only work if SmartFrog has built, network access is up and running, and the `core/smartfrog` build published the artifacts to the repository. It also runs the tasks' tests.

Running the testharness

Next, the testharness can be run.

```
>cd core/testharness
>ant all
```

This triggers a build of the testharness (which is published) and a run against all components which have been copied to `smartfrog/dist/lib` and whose test cases are in the testharness source tree.

> *Do not panic if one or two tests fail. It may be that way on other people's systems too!*

Some of our tests have been known to fail, and stay that way until the underlying problem has been identified. Therefore it is not unlikely that you may have checked out a copy of the source tree while a test was failing. Check with the developer mailing list to see if this is the case.

The complete process for testing SmartFrog is covered in the testing documents. What is useful to know is that, in the testharness project:

- the `buildtest` target rebuilds `../smartfrog`, then runs the tests, in one single go
- To run a single test, set the name of the test case class to run in the `testcase` property, with a `-Dtestcase=org.smartfrog.example.TestCase` on the command line

Although many components do have their test cases in this package, adding tests for new components here (other than those in `core/smartfrog`), is strongly discouraged. It is better for each component's tests to be in the component's own source tree, where they can be tested within a self-contained build.

Publishing

Now try publishing every artifact

```
>cd core
>ant published
```

This delegates to Ivy, which builds all components matching its pattern (excluding those excluded because they depend upon a later version of Java), publishing each artifact to the local Ivy cache (`~/.ivy/cache`) as it goes along. The order to build projects is determined from the individual `ivy.xml` files.

6 Ivy

The build uses Ivy for the following activities

- Downloading dependent libraries from the public repositories
- Publishing the JAR files built by individual projects to a shared location on the file system
- Publishing the source, javadoc and documentation Zip files for individual components.
- Retrieving artifacts published by other projects within the SmartFrog build tree.

It therefore has two main roles: one, it pulls in needed libraries from outside. Two, it is the way that different parts of the build share artifacts with each each other. Ivy is essential to the build, which is why the first complete builds need network access from Ant.

You cannon build SmartFrog without Ivy.

Ivy is currently under incubation at Apache, on the path to becoming a fully fledged project. The best coverage of Ivy is the ApacheCon EU 2007 slides and chapter 11 of Ant in Action. The latter covers Ivy 1.4.1, which is what the build (in March 2007) uses. This is the last pre-Apache release; there has been no Apache release yet.

We are moderately confident that a future release of ivy will be compatible with SmartFrog; we build under Apache Gump for regression testing against such tools. However, we have not yet migrated SmartFrog to an alpha release of the new 1.5/2.0 release, as there are less stability guarantees.

Changing versions of a library

Ivy makes it very easy to change a library version. Although you can hard code a version number into an `ivy.xml` file, we have instead chosen to keep all library version numbers in a set of properties, identified by `${project.version}`, where *project* can be a project such as log4j, JUnit, Xom.

The file `core/components/libraries.properties` defines the version properties of all JAR files used by all components in the SVN repository.

To add a new library definition, insert it in to this file *in alphabetical order*. The use of alphabetical ordering prevents duplicates, and makes it easier to merge in changes. It also makes it easier to find which version of library is in use.

To change a library version for all builds, edit the file. To change version on a single machine, edit the `build.properties` or `core/master.properties` files, defining the new version before `libraries.properties` gets read. Alternatively run Ant with the new definition on the command line:

```
ant clean published -Dlog4.version=1.2.12
```

Always do a clean build after changing a version number, as Ivy caches a lot of information in the `dist/` directories that needs to be deleted.

7 Setting up a Continuous Integration Server

SmartFrog has been built using Cruise Control, Luntbuild, Atlassian Bamboo and Apache Gump.

Here is some advice when setting up a Continuous Integration (CI) server to build SmartFrog

- Do not run the CI server on a developer's system, as functional testing interferes with any development builds, and Ivy can pick up the wrong artifacts. Use a Xen or VMWare image, or host on a different

physical machines.

- Get everything working on the command line first.
- Do not mix SVN 1.4 and SVN 1,3 clients, as they use different file formats. Java-based CI tools tend to use (in March 2007), the version 1.3 file formats, though Bamboo is an SVN 1.4 tool.
- Don't run different builds in parallel, if they are producing or consuming shared artifacts through the Ivy local repository, or running functional tests.

CI Proxy Setup

Configuring proxy information is one of the hardest problems. What appears to work well is the following:

1. Edit the batch file/shell script of the CI server to set up the proxies for the (Java-based program)
2. If you can configure which Ant is run, try and set the `ANT_OPTS` environment variable, or the `-D` settings for the proxy. These must be set for Java itself, and not Ant properties, which have no influence on proxy configuration.
3. If the program calls `ant.bat` or `ant.sh` to run Ant, edit this batch file/shell script to ensure that the proxy settings are always passed down.

8 Troubleshooting

Here are some common problems

Unsupported Ant Version

You must be running Ant1.7. If you want to build things under an IDE, make sure the IDE is pointed at the version that you have installed.

Proxy settings not picked up by Ant/Ivy, especially under an IDE or CI server

You can test this with the diagnostics target:

```
<target name="diagnostics" depends="init"
  description="build file diagnostics">
  <diagnostics/>
</target>
```

This target runs under the IDE/CI server, so shows exactly what Ant believes the situation is.

Ivy complains of unresolved dependencies

This means that it could not locate an artifact in the .ivy/local directory of published artifacts, or the external Ivy server. If it is an external artifact, check that the version is actually available on the public repository. We use MvnRepository for such searches.

If it is a local artifact, that is, one of the SmartFrog artifacts, then the dependent project has not published its artifacts. Change to the directory of the specific component/module, and run `ant published` in this directory.

Ivy complains of an "unknown configuration" of a dependency

This is a more serious problem. One project is asking for a specific *configuration*, a list of artifacts and dependencies from another module, but the named configuration is not known.

This may happen if a new configuration is added, but the component has not been rebuild/republished. Do an `ant clean published` on the identified component. The other cause is there is an error in either of the `ivy.xml` files.

Ivy complains of an "missing artifact" of a dependency

This is more serious; it means that a project did not publish an artifact that was expected. This could be due to an error in the project's `ivy.xml` file, or it could be that the build did not publish the artifact.

Out of date artifacts being picked up in the build

Everything that is built is published by Ivy to `~/.ivy/local` ; delete this directory to remove the locally created artifacts. Delete the entire cache under `~/.ivy/cache` to remove all artifacts downloaded by Ivy, and all cached metadata. There are some special targets in the main build file to trigger cleanup

Target	*Action*
`ivy-purge`	purge the cache and published artifacts.
`ivy-purge-published`	Delete all locally published artifacts from .ivy/local
`ivy-purge-cache`	purge all SmartFrog artifacts from the cache,

Out of memory during a build, particularly OutOfPermGenSpace

This appears to be caused by memory leaks of classes and/or classloaders in Ant across very big builds. To increaase the PermGenSpace, add the option to ANT_OPTS:

```
ANT_OPTS= -XX:MaxPermSize=128m
```

See http://jira.smartfrog.org/jira/browse/SFOS-167 for more details

Out of memory during a build inside <junitreport>

This can arise on Java1.5+ when the built in XSLT engine (xsltc) is used to process the reports. Download and add Apache Xalan to your ANT_HOME/lib and all should be well. Alternatively, extend the heap space in Ant through ANT_OPTS

```
ANT_OPTS= -Xmx512m
```

Tests failing

These may be failing for reasons unrelated to your activities. Check with the team and the CI server(s) to see if they are known faults.