IDE Development

20 Jul 2007

1 Introduction

This document covers how to work with the SmartFrog source trees from inside IDEs.

2 Running SmartFrog under an IDE

2.1.1 Before using the IDE

- Run "`ant published`" in the core directory, to build smartfrog.jar, the Ant tasks, and all non-experimental components. It is essential to build the main distribution in Ant, at least initially, to create Java source files matching the language grammar. The Ant tasks are mandatory for the testharness and component development.
- Run `sfDaemon` to bring up the standalone daemon. This helps isolate problems; If something does not run standalone, it will not work under an IDE either.

2.2 Starting SmartFrog daemon under an IDE

If you start a daemon under the IDE, it can listen for incoming deployment requests, such as those coming from a test case, deploying the relevant components on demand.

1. Set up the classpath to include that of all components, or write a deployment descriptor that includes the path to needed JARs as file: URLs
2. Create an executable entry point "`org.smartfrog.SFSystem`"
3. Set the following JVM property:
   `-Dorg.smartfrog.sfcore.processcompound.sfProcessName=rootProcess`
   This tells SmartFrog to export the root process for incoming deployment requests. It is the key to turn SmartFrog into a daemon.
4. Set the following (optional) JVM options
   `-Dorg.smartfrog.logger.logStackTrace=true`
   `-Dorg.smartfrog.iniFile=bin/default.ini`
   This turns stack tracing on in the logs; the second points to any initialisation file to load.
5. Set the `-ea -esa` options if you want any assertions in components to be enabled
6. Set breakpoints on any code/data you wish to debug
7. Run the test case or deployment of choice. This can be from the command line, or from the IDE itself.

Security Advisory: When running a daemon under the IDE, don't open the port of the daemon to the rest of your network; don't allow anyone other than yourself a login on the localhost.

2.3 Running a SmartFrog deployment descriptor under an IDE

This is the easiest way to debug a deployment. Instead of starting a (potentially insecure) daemon, you run a local deployment under the debugger.

1. Set up the classpath to include that of all components, or write a deployment descriptor that includes the path to needed JARs as file: URLs
2. Create an executable entry point "`org.smartfrog.SFSystem`"
3. Set the following (optional) JVM options
   `-Dorg.smartfrog.logger.logStackTrace=true`
   `-Dorg.smartfrog.iniFile=bin/default.ini`
   This turns stack tracing on in the logs; the second points to any initialisation file to load.
4. Set the `-ea -esa` options if you want any assertions in components to be enabled

5. Set breakpoints on any code/data you wish to debug
6. Identify the descriptor you want to deploy, with the following command line entries
   `-a :DEPLOY:files/testDeployment.sf:::`
   `-e`

This deploys the file `"./files/testDeployment.sf"`. You will need to set the home directory of the program up correctly, or use an absolute path. The `-e` argument tells SmartFrog to exit after the deployment.

2.3.1 Running Unit tests under an IDE

If the IDE can run JUnit test suites/packages, configure it to run the JUnit tests of the testharness or component of choice, *while running the daemon*. The daemon can be hosted in the debugger, or running standalone. Unless the test deployment configurations declare the `sfCodebase` of downloadable JAR files, all the JAR files that the component needs must be already in the JVM of the daemon.

The JUnit tests need to know the location of the directory into which the test classes have been compiled. This is so that it can construct file: URIs to the deployment descriptors in the class tree. As the Ant `<junit>` test does, the JUnit instance must have the the system property `test.smartfrog.classesdir` set to the absolute location of `testharness/build/test/classes`

```
-ea -esa
-Dtest.smartfrog.classesdir=/home/user/core/testharness/build/test/classes
```

2.4 Running Ant builds under the IDE

As modern IDEs all host Ant, you should be able to run the Ant build files directly from the IDE. Here are some caveats.

- The IDE's version of Ant may be older than that which the build files or SmartFrog Ant tasks need.
- You may not have all the optional JARs, such as JUnit or Xalan, in the IDE's own copy of Ant. These are needed for `<junit>` and `<junitreport>` respectively. If you can get the IDE to run against your existing installation, things are better.
- The IDE Ant engine may not autoload JAR files in `${user.home}/.ant/lib.`
- The build can be (noticeably) slower. This appears to be related to reporting.

2.5 Running Ant test suites under the IDE

The testharness/test target runs the SmartFrog system tests; some other projects have self-contained test suites. Usually you want to use the IDE to run a single test case/package, rather than the full system. Run the testharness build file, with:-

1. The testcase property bound to the test class you want to test
   `-Dtestcase=org.smartfrog.test.system.java.LibraryTest`
2. Use the target buildtest to compile smartfrog then compile and run the test suites, generating the HTML reports on failure
3. Use the target buildtest-noreports to omit the XSL transforms that create the HTML pages. This can save time when you are rerunning tests, and can read the failure messages that get printed to the console.

3 IntelliJ IDEA

IntelliJ IDEA works very well as the IDE for coding SmartFrog and components. As it supports multiple simultaneous debug sessions, you can debug the daemon and a unit test at the same time. It also lets you use an external version of Ant, which eliminates discrepancies between the IDE version of Ant and the command line version.

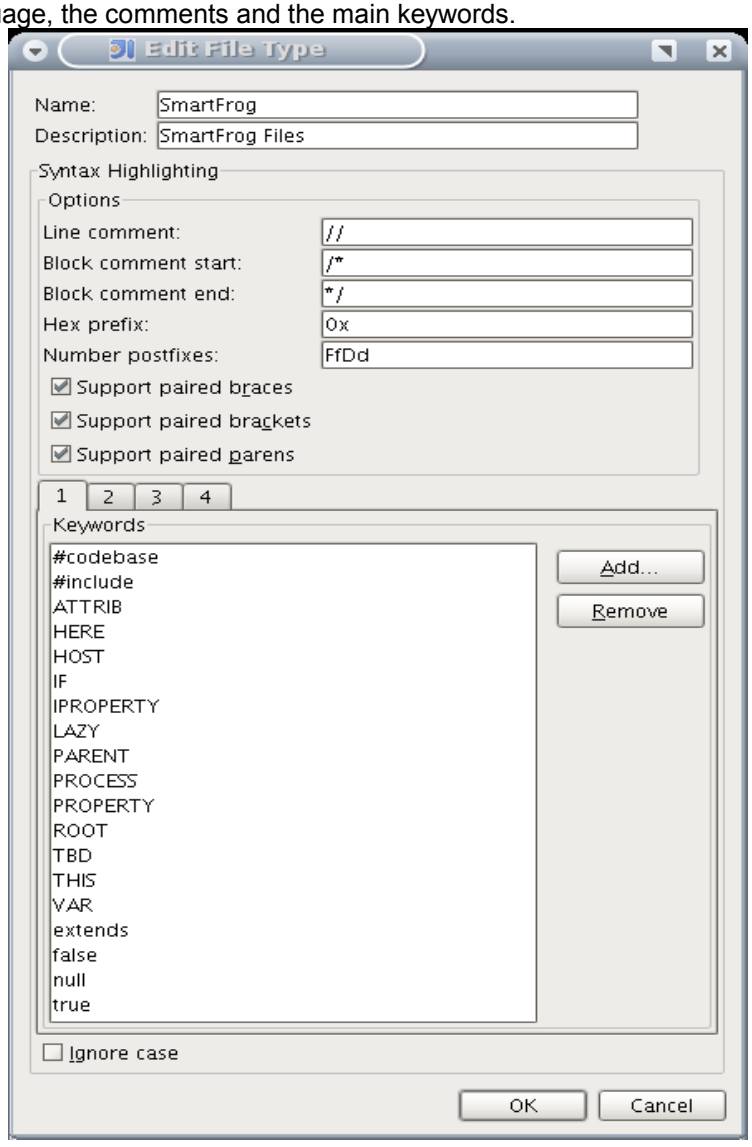
1. Create an empty project.
2. Add sub projects for smartfrog, testharness, and any components you wish to build/use.
3. Configure SVN if you have write access to the repository.

3.1.1.1 Tips

- Beware of refactoring code that may be used in components. Only those components currently in the set of projects will be refactored; external components will not pick up changes.
- Beware of auto-optimising imports as it can change the import layout quite drastically, especially if your system is set up to turn more than three imports into an import. *

3.1.1.2 Setting up a new SmartFrog filetype

The best way to edit SF files inside IDEA is to create a new file type. This can be bound to .sf files. The basic setup declares the language, the comments and the main keywords.



1. The second keyword tab can be set to a list of common components and attributes. For example, here is the list of attributes defined in prim.sf, each of which can be added:

```
sfAssertionPhase
sfClass
sfCodeBase
sfDeployerClass
sfExport
sfExportPort
sfLivenessDelay
sfLivenessFactor
sfLog
sfProcessName
sfProcessComponentName
sfProcessHost
sfShouldTerminateQuietly
sfShouldDetach
sfUpdatable
```

3.2 Eclipse 3.1

Eclipse 3.1M6 has been used to develop some SmartFrog source.

3.2.1.1 Tips

- Configure the workspace to autosave everything before running Ant, and when working with CVS.
- After doing major configuration changes, restart the app so as to force saving of all your settings.

3.2.1.2 Ant Integration

We have encountered problems here. Problems such as:

```
compile-source:
     [echo] 1.4 build.compiler=modern javac.compiler=modern
compile-rmi:
     [rmic] RMI Compiling 49 classes to core/smartfrog/build

BUILD FAILED
java.lang.UnsupportedClassVersionError: sun/rmi/rmic/Main (Unsupported
major.minor version 49.0)
```

This was tracked down to Java version problems, with a Java1.4 runtime and a Java1.5 SDK mix. Interestingly, the same problem did not arise on the command line. The error was not in the IDE itself, it merely triggered the problem. Uninstalling the Java1.4 JRE corrected things.

The other issue is that with a different Ant runtime, optional files such as junit.jar and xalan.jar were not on the classpath. You have to edit the build.xml properties to set ANT_HOME to that of your ANT runtime. This will give you the Ant and extra JARs of your command line.

3.2.1.3 SVN Integration

If not already in the file, add the following to the SVN Ignore list

```
build.properties
.classpath
.project
```

This stops team synchronization trying to add personal files to the repository. We have found that having a `.svignore` file at the base of every project/component directory stops build directories from being synchronized. In the SmartFrog project itself, care must be taken not to add machine-generated source to the repository; there is a special `.svngnore` in the relevant directory to prevent this.

Overall, we currently find the 'seamless' synchronization feature of Eclipse, namely its willingness to add all

unmanaged files to the repository *unless explicitly included,* somewhat overaggressive. The biggest problem is that CVS server handles deleted directories very badly; if someone were ever to accidentally import the build/ directory of a component, nobody would be able to rebuild that component until the SourceForge support team fielded a "please delete this directory from CVS" request.

To disconnect a project from CVS, bring up the context menu on a project in the navigator, then select Team| CVS|disconnect, then confirm the disconnect, making sure that the “do not delete” option is set, so that other CVS tools work normally.

4 jEdit 4.2

There is a XML syntax file describing .sf files, in the smartfrog/doc directory. Add this to jedit/modes then edit modes/catalog

```
<MODE NAME="smartfrog" FILE="jedit-smartfrog.xml"
        FILE_NAME_GLOB="*.sf"/>
```

After restarting jEdit, .sf files will be given syntax highlighting.

4.1 Dependent Projects

Many IDEs work best if the source for dependent libraries and tools are available. We recommend having a local copy of:

- JUnit - for Unit test debugging
- Ant source tree -if you are working with the Ant tasks.
- Dependent libraries of components which you are developing.

4.2 Subversion Integration

Developers with write access to the SourceForge repository can enable SVN integration, if their IDE supports it.

- Make sure that core/.svnignore is being processed, and that the IDE is not trying to aggressively synchronize all output in the build directory (as Eclipse Team Synchronization is wont to do).
- Do an update, clean build and test before checking in anything. This will reduce the risk of inadvertent collisions.
- Do not check in broken code, otherwise the CI builds will complain.
- Remember to check in files such as rmitargets, and core/common.xml, if they have changed.
- There is a mailing list just for checkins. We encourage all active developers to watch this email.

5 Things to Watch out for

- Writing code specific to your version of Java. This can be Java1.5 framework features, or it can be using "`enum`" as a variable name in Java1.4 applications.
- Writing code against the build of a library set up with the IDE, not that in the lib dir of a component.
- Gratuitous re-ordering of imports, comment layout, etc, etc. These confuse CVS and create conflicts when there should be none.
- Not maintaining the build file of a component. The build file is the way to build components in production; IDE-only builds are for development only.
- Changing the public API by accident, while refactoring.
- Checking in code having only verified that a single test case/package works not the whole testharness.

- Import handling replacing explicit imports with .* imports. Set the number of imports in a package to trigger this feature in an IDE to 999 to effectively disable it.