JUnit Components – enabling distributed Unit Testing

This set of components can run JUnit test cases on arbitrary machines in the system. As such it enables remote testing in a distributed network, and the ability to test code running on different platforms.

Tests can be run on different machines, with the results collected on a single system in XML form. Post-processing can then create human-readable reports.

Architecture

1. The component will load and run multiple JUnit `TestSuite` or `TestCase` instances, instances specified by the user (along with the classpath that contains them).
2. The test cases will initially run in the SmartFrog JVM; we may want to add forking later.
3. The output of the test cases will be reported to any registered listeners,
4. We provide two listeners: a console listener (that routes to the current `System.out`), and an `XMLListener` that saves its state to an XML file that is compatible with the Ant format.
5. We may also want to pull in the Ant reporter, or at least directly invoke the Ant code from SmartFrog; the latter keeps our engineering costs down.

Components

The schemas for these components are defined in

`/org/smartfrog/junit/components.sf`

The classes are implemented in `sf-junit.jar`; this and `junit.jar` itself must be on the classpath of the deployment descriptor running the tests. Note that the latest released version of JUnit was used when building these components; at the time that was JUnit 3.8.1.

TestRunner

The `TestRunner` is the core component, it runs test suites declared inside it, and forwards the results to a nominated listener.

The runner will be configured by multiple `TestSuite` declarations inside the declaration.

`listener`: reference	name of something that listens to the tests
`keepgoing`: boolean	optional, true by default. Tells the system to keep going on the failure of a single test. If false, the first test to fail will stop the test suites from continuing.
`failOnError`: boolean	optional, default true. Fail if a test failed.

The class will start running the tests when deployed; when it is finished it will terminate itself. It thus remains alive for the duration of the tests.

TestSuite

A `TestSuite` implementation is a component that knows how to run a suite of tests. The only current implementation is the `JUnitTestSuite`, which runs JUnit tests. The reason for having a base class that is not JUnit-aware is that it allows new components to be written which run different test frameworks.

JUnitTestSuite

A JUnit Test Suite runs JUnit tests.

`classes: String list`	list of full names of classes implementing `junit.TestSuite` or `junit.TestCase`
`if: boolean`	conditionally execute the test only when `if==true`
`unless: boolean`	conditionally execute the test only when `unless==false`
`package`	name of a package that contains multiple test cases/classes.

When a package is named, all classes in the package that are not abstract, and which implement `junit.framework.TestSuite` or `junit.framework.TestCase` will be executed. This is much simpler than Ant's pattern based selection mechanism, which permits recursive declarations such as "`**/*Test.class`".

It may also be useful for a `TestSuite` to be recursive; a `TestSuite` can contain multiple subsidiary `TestSuite` components.

TestListenerFactory

This interface provides instances of `TestListener` implementations. Callers should ask the factory for an implementation of the interface, passing in the hostname, and suite name. The factory will return an interface instance to use for test reporting.

TODO: add process name too. This helps us isolate tests running in parallel processes, such as those running against different endpoints.

TestListener

This is an interface for reporting tests. All the normal JUnit callbacks are exported, with a serializable `TestInfo` parameter. The `endSuite()` method *must* be called at the end of the test suite, after which the interface methods *must not* be called again. The method exists to close files and do other cleanup. We have deliberately not chosen to rely on the class finalizer to perform such operations as we do not know if or when it gets called -especially once distributed garbage collection gets involved.

XMLTestListener extends TestListener

The `XMLTestListener` listener generates XML log files. These are not the same as the Ant files, because the Ant implementation has to buffer all tests in memory until finished, so that the results can be summarised in attributes of the root node.

The XML schema used (and it *is* a schema) is designed to save such summary information to the end of the document, so that test results can be streamed out to the file as it proceeds.

Log information can be embedded inside the document.

ConsoleListener extends TestListenerFactory

The Console Test Listener streams test results to an output stream of choice; usually the console. Note that `System.out` will not work once the tests start running; output to that stream will be diverted into the test results themselves.

BufferingListener extends TestListenerFactory

The implementation of this component saves all results to lists inside itself. It exists purely for testing the framework itself. JUnit tests can deploy a test suite then make assertions about the state of the log.

Example

Here is the use made in testing the SOAP endpoints of the CDDLM implementation. We define a test base template that first has an undefined endpoint, which is passed to JUnit as a property. Then comes three separate templates, one for each package, listing the tests in each one.

```
#include "/org/smartfrog/services/junit/components.sf"
#include "/org/smartfrog/services/filesystem/components.sf"

/*
 This is a package of tests that run against a local or remote server
 that is served up by the endpoint specified.
 */

  AlpineTestSuite extends JUnitTestSuite {
        name "alpine";
        enabled true;
        if enabled;
        //full path to the echo endpoint
        endpoint TBD;
        properties [
          ["endpoint",LAZY endpoint]
        ];
  }


  PortalTestSuite extends AlpineTestSuite {
        name "portal";
        package "org.smartfrog.services.deployapi.test.system.alpine.deployapi.api.portal";
        classes [
          "Api_01_portal_wsdm_properties_Test",
          "Api_02_portal_getproperty_unknown_Test",
          "Api_03_portal_getproperty_unknown_prefix_Test",
          "Api_04_portal_getPortalState_Test"
          ];
    }

  SingleTestSuite extends PortalTestSuite {
        classes [
          "Api_01_portal_wsdm_properties_Test"
          ];
  }

  CreationTestSuite extends AlpineTestSuite {
        name "creation";
        package "org.smartfrog.services.deployapi.test.system.alpine.deployapi.api.creation";
        classes [
          "Api_05_system_create_destroy_Test",
          "Api_06_system_properties_Test",
          "Api_07_system_destroy_properties_Test",
          "Api_08_system_destroy_operations_Test",
          "Api_09_system_active_systems_Test",
          "Api_10_system_on_invalid_hostname_Test",
          "Api_11_system_on_valid_hostname_Test",
          "Api_13_system_multiple_systems_Test",
          "Api_14_system_lookup_Test",
          "Api_14a_system_destroy_lookup_Test"
```

```
            ];
    }
```

Implementation

The first non-forking implementation runs tests using the same classpath used to deploy the components. Each test suite can be deployed on a different host from the `TestRunner`, and with a different classpath.

The tests begin run when the component's `sfStart()` method is called; the tests are run in a separate thread and will continue until finished. If the component is terminated the test thread will be interrupted and so exit when the next test class has finished.

After the tests have run, if any test failed (or there was an error such as a `RemotingError`) during execution of the tests, the next liveness check on the `TestRunner` will signal a fault. If the `failOnError` parameter of the `TestRunner` is set to false, this does not happen.

Issues

Remoting

We did not attempt to remote the normal version of `junit.framework.TestListener`, because it is fundamentally unremotable. The key issue is that an arbitrary `Throwable` can be included in a failure, so serialization is out of the question. Nor do we have the ability to mark existing JUnit classes as `Serializable`.

This means that we have a serializable form of test case results, `TestInfo`, which must also serialize and nest exceptions. The latter is done by extracting salient information from the test (message, classname and stack trace). All other exception information is lost.

The second revision of the serializable classes added the ability to include log data inside the test information, because this integrates logging into the results. In use, it soon became clear that log information aided diagnosis of the underlying cause of test failures.

Forking

To fork, the caller must declare the `sfProcessName` attribute of the `TestRunner` or `TestSuite`. SmartFrog will then automatically run the tests in a process of that name, creating it if it does not already exist.

Package Searching

Of course, it would be too easy for the Java Classloader API to provide a way to enumerate all classes in a package. There appears to be no way to do this. The first pass implementation therefore requires all test cases/suites to be named explicitly.

Future workarounds may be

1. Generation of the list at compile time, from an Ant `<fileset>`
2. Generation of the list by looking inside a jar file with the `java.utils.zip` API

This is pretty tricky. For now, the JUnitTestSuite component takes a package name and a list of classes; these are appended to the package name to create the list of classes to test.

Timeouts

Tests hang. Those constitute failures of a drastic kind. The normal JUnit runner does not appear to handle timeouts, but the Ant one does, by setting a timeout in the CommandLine Java class. This works OK, if you fork, at the price of leaving an empty XML file around. If a non-forked JUnit test is killed, your system is left in an indeterminate state, as neither Java nor the OS clean up completely.

Ideally, we'd like the testsuites and testrunners to be timeout aware, with per-suite and per-test timeouts. The first implementation has chosen a different path, implementing a new compound class, TimeoutCompound. This class takes a timeout (in seconds) and will terminate all unterminated children when the timeout is reached and they are still unterminated.

While generically useful, it is not as intimate as timeout aware test runners, because they can log the timeout event to the test listener, then clean up as much of their state as they can.

Testing the test framework

This is an interesting problem. How do you test the test framework? Here is the current plan:

1. Write dummy tests cases that always succeed, fail or throw an exception
2. Write deployment descriptors that run the different test cases, all reporting to the `BufferingListener`
3. Write JUnit tests that extend `SmartFrogTestBase` and can deploy descriptors to a running daemon. These tests make assertions of the buffering listener.
4. Host the outer tests in Ant.

This tests local functionality. Distribution requires us to be hosting SmartFrog daemons on the remote machines; the tests themselves should function as before.

Configuring Tests

How to configure tests?

Reporting

Now that we are capturing results and log data, we need to think how to report it. Here are options.

- Generation of HTML pages for a static web site from the XML, by way of XSLT. This what Ant's <junitreport> task does.
- Forwarding data to a client-side GUI application, as with GridUnit
- Generating and forwarding result email to developers
- Saving the content to some form of database (relational or XML), then having a dynamic web application provide access to the database. This would enable many fancy features, such as tracking of changes over time, viewing along different axes (view by host, by test, by options). It would also result in the most complex web application.