

JSS (Java Serialized Stream) Overview

Java Serialized Stream Extension for Rational Suite TestStudio

An adaptor for testing applications that use Java Objects for communication between clients and servers.

The JSS Extension transforms raw socket data in previously recorded test scripts into lines that separate and identify the data elements of the Java Objects being transferred. The newly formed scripts can then be used to perform load, stress, and performance testing and benchmarking against the application.

Recording Java Serialized Streams

The Rational TestSuite product line does not have support for the Java Object Serialization protocol, which will cause recordings on applications that use this protocol to have fragments similar to this simple example:

```
testhost_1 = sock_connect("identifier", "testhost:5678");
set Server_connection = testhost_1;
sock_send
  "aced0005`sr`000c`AllPrimitive`c73a9c3cb013e42902000943000b`anotherChar"
  "Z`0007`theBoolB`0007`theByteC`0007`theCharD`0009`theDoubleF`0008`theFlo"
  "atI`0006`theIntJ`0007`theLongS`0008`theShortxp`004b011200`J?`f3c0ca42d8"
  "b3a93f9e06`RI`9602d2002bdc`T]kk`87`09";
sock_recv ["identifier"] "$"; /* 123 bytes */
sock_disconnect(testhost_1);
```

In order to use a script fragment like this in a performance test run, one usually replaces several key data elements with elements from a datapool. However, in this case, it is rather cumbersome to identify the data elements in the object, and even harder to come up with a VU statement that allows one to extract data.

After running this script fragment through the JSS extension, it will look like this:

```
testhost_1 = sock_connect("identifier", "testhost:5678");
set Server_connection = testhost_1;
sock_send
  jss_begin( "5" ) +
  jss_new( _object, "obj1", "" ) +
  jss_def( _class, "obj1.<name>", "AllPrimitive" ) +
  jss_def( _class, "obj1.<uid>", "c73a9c3cb013e429" ) +
  jss_def( _class, "obj1.<flags>", "SC_SERIALIZABLE" ) +
  jss_def( _class, "obj1.<num_fields>", "9" ) +
  jss_def( _field_p, "obj1.anotherChar", "char" ) +
  jss_def( _field_p, "obj1.theBool", "boolean" ) +
  jss_def( _field_p, "obj1.theByte", "byte" ) +
  jss_def( _field_p, "obj1.theChar", "char" ) +
  jss_def( _field_p, "obj1.theDouble", "double" ) +
  jss_def( _field_p, "obj1.theFloat", "float" ) +
  jss_def( _field_p, "obj1.theInt", "integer" ) +
  jss_def( _field_p, "obj1.theLong", "long" ) +
  jss_def( _field_p, "obj1.theShort", "short" ) +
  jss_raw( "x" ) +
  jss_def( _par_cls, "obj1", "" ) +
  jss_data( _char, "obj1.anotherChar", "K" ) +
  jss_data( _bool, "obj1.theBool", "TRUE" ) +
  jss_data( _byte, "obj1.theByte", "0x12" ) +
  jss_data( _char, "obj1.theChar", "J" ) +
  jss_data( _double, "obj1.theDouble", "1.2345678912345" ) +
  jss_data( _float, "obj1.theFloat", "1.2345679" ) +
  jss_data( _int, "obj1.theInt", "1234567890" ) +
  jss_data( _long, "obj1.theLong", "0x2bdc545d6b4b87" ) +
  jss_data( _short, "obj1.theShort", "12345" ) +
  jss_end();
sock_recv ["identifier"] "$"; /* 123 bytes */
sock_disconnect(testhost_1);
```

Note that in this example, the raw data is split up into data and structural elements. The `java_data()` lines identify the type, field name, and value of each of the data elements, the `java_def()` lines show structural definition information about the objects, and the `java_raw()` lines are all the elements that are either not interesting or not supported yet.

Playing back Java Serialized Streams

Playing back scripts as shown above is straightforward. The `java_data()` and `java_raw()` commands translate their arguments back into the mixedmode form of the original script, only this time the user has replaced key data elements by values from a datapool.

Retrieving data elements from the incoming streams is also quite easy. By causing Robot to generate dumps of the incoming object streams (which the JSS also translates for you), one can look up the name for a certain element within a received object, and then simply call the `jss_get_data()` function to retrieve its data:

```
my_data = jss_get_data( "obj1.theInt" );
```

which would retrieve the value of the data element called "theInt" (which is "1234567890" in the above example). The returned string value can be used in further calls `java_data()` commands, allowing for easy including of session identifiers in your test scripts.

Supported Versions of Rational

We support the following versions:

- IBM Rational TestStudio, TestManager, Robot, or PerformanceTester
- v2003.06.12 or v2002 patch 7 (or above)

Supported Platforms

We support the following platforms as noted.

<i>Operating System</i>	<i>Feature</i>	Script generation	Script playback
Windows NT 4.0 SP6a		Yes	Yes
Windows 2000 Professional, Server, and Advanced Server with SP3		Yes	Yes
Windows XP Professional with SP1		Yes	Yes
Windows 2003 Server		Untested	Untested
Linux (testing performed on RedHat)		n/a	Upon request
HPUX 10.x,		n/a	Upon request
HPUX 11.0 and 11i		n/a	Upon request
IBM AIX		n/a	Upon request
Sun Solaris		n/a	Upon request

Supported Versions of Serialization

We support the following versions of the Java Object Serialization Protocol

- 1.0
- newer versions not yet released

JSS Product Features

The following is a highlight of product features and level of TestStudio integration

- Works with native Rational TestSuite scripts
- Automated installation of the adaptor
(working on) Custom interface to control filter, Generate, and Replay Options, integrated in Robot
- Filter for Session File supported; protocol displayed in Generator:Filtering tab
- Script re-generator supports split scripts, comments, and timers
- Scripts generated in standard VU language, platform independent
- Scripts can be compiled and replayed without further modification
- (in test) Replay libraries automatically handle platform byte-ordering differences.
- Replay logging available via TestManager

For more information about the JSS Extension, send an e-mail to: info@zyntax.com specifying your interest in the JSS product.