Extensibility and Scripting Basics

This topic provides an overview of issues related to SOAtest' s and Virtualize's extensibility (scripting) capabilities and their various applications. Sections include:

Understanding the Extensibility Feature

The SOAtest & Virtualize extensibility feature allows you to apply custom scripts (using Java, JavaScript, Oracle Nashorn, Groovy, or Jython) to perform any specific function that would be helpful to you while working with SOAtest & Virtualize. It also supports other scripting engines that implement the JSR 223 "Scripting for the Java Platform" specification. This feature gives you the ability to customize SOAtest & Virtualize to your specific needs without learning an application-specific language.

If you work with SOAtest, you can also use scripting to customize rules you create with RuleWizard. For details on this feature, see the RuleWizard User's Guide.

Setting Your Environment for Scripting

If you are using Jython scripts, you might need to specify your jython.home and jython.path variables in the Scripting tab of the Preferences panel. Both variables are used to locate Jython modules, and Jython code that does not import any Jython modules can use the Jython scripting support without setting either variable. jython.home specifies the Jython installation directory. jython.path is used to add to your path modules that are not in your jython. home/Lib directory. Multiple paths can be listed in jython.path, while jython.home must be a single directory. If you set the jython.home and jython.path variables, you need to restart SOAtest or Virtualize before the changes will take effect.

If you are using Java for scripting and want to use SOAtest recompile modified Java files, see Using Eclipse Java Projects in SOAtest for details on how to set your environment on scripting.

If you are using Java for scripting and want to use Virtualize to recompile modified Java files, see Using Eclipse Java Projects in Virtualize for details on how to set your environment on scripting.

Specifying the Script

To define a script from SOAtest or Virtualize GUI controls that provide a scripting option:

1. If your Java class is from a Java project in your Eclipse workspace, add the Java project to the classpath.
2. From the **Language** box, indicate the language that the script will use.
3. Define the script to be implemented in the large text field.
    - For Java methods, specify the appropriate class in the **Class** field. Note that the class you choose must be on your classpath (you can click the **Modify Classpath** link then specify it in the displayed Preferences page). Click **Reload Class** if you want to reload the class after modifying and compiling the Java file.
    - For other scripts, you can use an existing file as the source of code for your method or you can create the method within SOAtest or Virtualize.
        - To use an existing file, select the **File** radio button and click **Browse**. Select the file from the file chooser that opens, then click **O** to complete the selection.
        - To create the method from scratch from within SOAtest or Virtualize, select the **Text** radio button and type or cut and paste your code in the associated text window.
        - To check that the specified script is legal and runnable, right-click the **File** or **Text** text field (click whichever one you used to specify your script), then choose **Evaluate** from the shortcut menu. SOAtest or Virtualize will report any problems found.

Using Script Templates

If you create scripts using Jython or JavaScript, you can specify a script template in the SOAtest or Virtualize Preference panel's **Script Template** field (in the Scripting page). Whatever code is specified in this field will be used as the default code for inlined scripting in the language with which the field is associated. This is primarily useful for setting default inputs and common global variables.

Script templates apply to scripts used in Extension tools, SOAP services, and scripted SOAP inputs.

Interacting with SOAtest and Virtualize

If you need your scripts to interact with the SOAtest or Virtualize program, you can do so using the SOAtest or Virtualize Extensibility API. For example, you can use the SOAtest or Virtualize Extensibility API to send results of a script to a SOAtest or Virtualize Message window or pass specific values to specific SOAtest or Virtualize.

You can access documentation for the Extension framework API via the **Parasoft> Help** menu (look for the book titled "

Accessing Required Jar Files

If your Java code uses the Extensibility API and you want to compile the code within the

SOAtest

or

Virtualize

environment, use the Java Project wizard. This wizard will create a new Eclipse Java project that has access to the Extensibility API. For details, see

Using Eclipse Java Projects in SOAtest

or

Using Eclipse Java Projects in Virtualizefor details.

If you want to compile the code outside of the

SOAtest

or

Virtualize

environment, you will need the following jar files:

- com.parasoft.api.jar
- webking.jar (only required if you will be accessing classes from the soaptest.api or webking.api package)

These jar files are available at`<SOAtest Installation Directory>\plugins\com.parasoft.xtest.libs.web_<SOAtest version>\root` and`<Virtualize Installation Directory>\plugins\com.parasoft.xtest.libs.web_<Virtualize version>\root`

Building with Maven?

If you're building your Java project with Apache Maven, you can add the following to your project's pom.xml:

```
<project>
  ...
  <dependencies>
    <dependency>
                <groupId>com.parasoft.soavirt</groupId>
                <artifactId>com.parasoft.api</artifactId>
                <version>9.9.0</version>  <!-- should match product version -->
    </dependency>
  </dependencies>   ...
<repositories>
    <repository>
                <id>ParasoftMavenPublic</id>
                <name>Parasoft Public Repository</name>
                <url>http://build.parasoft.com/maven/</url>
    </repository>
  </repositories>
  ...
</project>
```

Additionally, if scripts need to use other functionality beyond these API classes, you may need to add additional (3rd party or open source) jars to your classpath. For example:

- For XPath processing, you would also need the class org.apache.xpath.XPathAPI. This is located in xalan.jar.
- For ISO 8583-related scripting, you would also need classes from org.jpos.iso.* such as ISOMsg, BaseChannel. ISOChannel, and ISOPackager. These are located in jpos.jar.

Setting the Context Classloader

The Extension tool runs in the classloaders. Certain APIs, such as JNDI, are sensitive to the context classloader. There may be cases where your code must explicitly set the context classloader before performing a particular API call. For more information, see http://wiki.eclipse.org/Context_Class_Loader_Enhancements#Technical_Solution.

Script Variables

You can declare and use variables in SOAtest and Virtualize scripts.

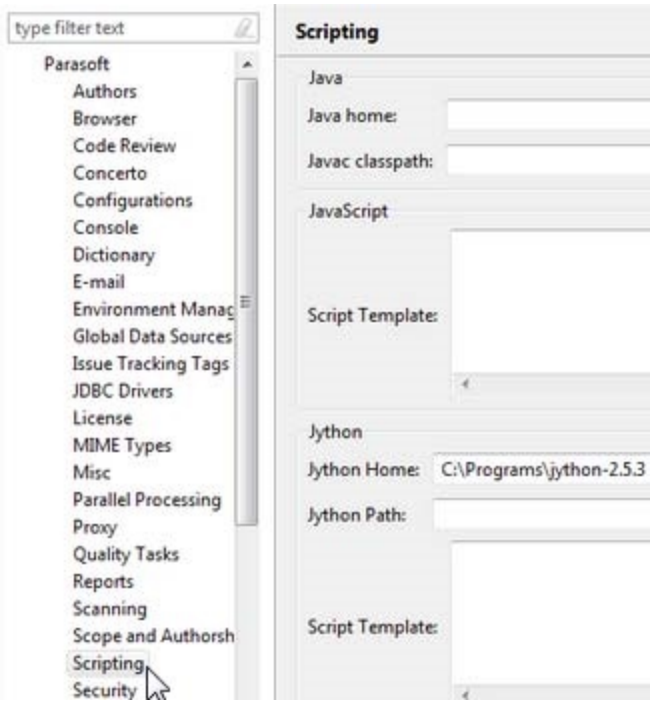
Variable declaration must start with a keyword `"var"` followed by an equals sign `"="` followed by a variable value. Each variable must be declared on a separate line. Variables can be declared anywhere in the script, a variable can be used immediately after its declaration. To assign an existing variable a new value, `"re-declare"` the variable.

A variable can be referenced in a script as follows: `${varName}`.

Configuring Jython Preferences
Jython Version

The Jython 2.5.2 version that ships with SOAtest and Virtualize does not include some of the standard Jython libraries, such as the os module. If you are used to using these modules and would like to access to them in SOAtest or Virtualize, you can point SOAtest or Virtualize to an external Jython installation as follows:

1. Install Jython in a location of your choice; we will refer to this location as `<Jython Install Directory>`.
2. Open SOAtest or Virtualize.
3. Choose **Parasoft> Preferences**.
4. Select **Scripting**.
5. In the **Jython Home** field, enter `<Jython Install Directory>`.



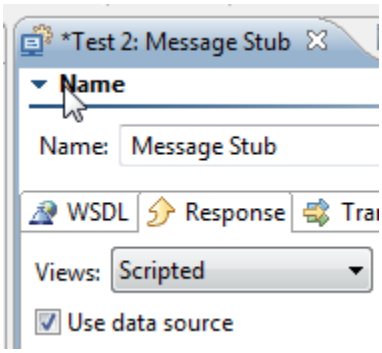
6. Restart SOAtest or Virtualize.

Jython Syntax Coloring

Jython syntax coloring for the Extension tool is enabled using the PyDev plugin (http://www.pydev.org/). This plugin must have its jython interpreter configured for advanced functionality such as code auto completion. For more details, see http://www.pydev.org/manual_101_interpreter.html.

Accessing Data Sources from Scripts

If you use SOAtest, you can make a data source available to a script by selecting the data source in the top of the tool's configuration panel, and then enabling the **Use Data Source** option.



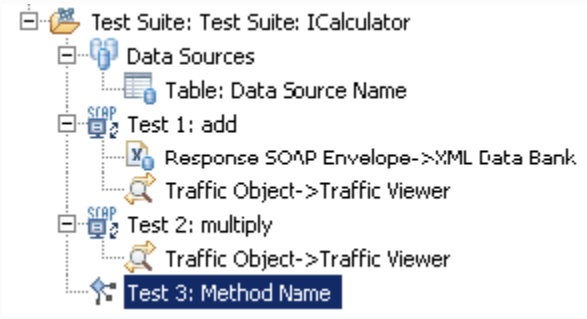
Within other methods in your script, or within any method in the script if you use Virtualize, store a value from the data source into a variable "x" with the line:

```
x = context.getValue("Data Source Name", "Column Name")
```

For more information on scripting, see the documentation for the Extension framework API. Choose **Parasoft> Help**, then look for the book titled "Parasoft SOAtest Extensibility API" (for SOAtest) or "Parasoft Virtualize Extensibility API" (for Virtualize).

SOAtest Example and Notes

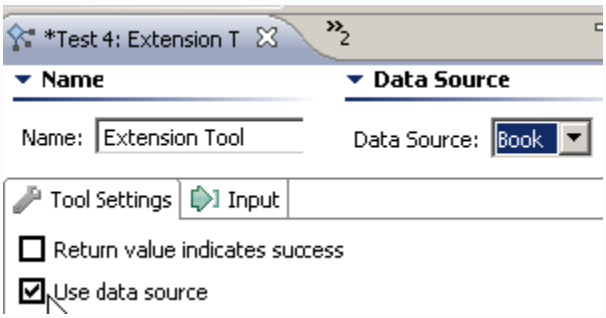
For example, let's say you have a test that looks similar to that in the following image:



The name of the table data source is **Data Source Name**. Not pictured are the following important facts:

- The table **Data Source Name** has a column labeled **Column Name**.
- The XML Data Bank contains a column labeled **Test 1: Result**.
- The XML Data Bank and Writable Data Source belong to a data source labeled **Generated Data Source**.

In order to make one of these data sources available to your script, select the data source in the top of the Extension tool's configuration panel, then check **Use Data Source**.



Notes

When opening older files (created prior to SOAtest 6.1), note that Browser Data Bank Tool column names are automatically transformed to "Extracted:xyz", where "xyz" was the column name that you specified. This provides support for legacy scripts that reference "Extracted:xyz". You can change the column name in the Browser Data Bank to just "xyz" or "abc."

For more information on scripting, see the documentation for the Extension framework API. Choose **Parasoft> Help**, then look for the book titled "Parasoft SOAtest Extensibility API".

Extensibility Examples for SOAtest

Accessing, Manipulating, and Storing Data

One typical usage of an Extension tool is accessing data from a data source, manipulating it, and then storing it dynamically in an XML Data Bank or Writable Data Source. In this scripting example, we define a method called getKeywords in which we are accessing data from a data source titled “Books” with a data source column titled “keywords”. We then return an XML representation of this string so that we can send the output from this script to an XML Data Bank.

Note that additional configuration is required to access Data Sources from an Extension Tool. See the previous section for details.

In these samples, the methods take two arguments as input:

- Object input: Represents input passed to the method from a previous test case. For example, if we chain an Extension tool to the SOAP Response of a SOAP Client tool, the input Object will be a string representing the SOAP Response.
- ExtensionToolContext context: The method getValue gives access to Data Sources from an Extension Tool. This method can be used along with other methods exposed by the Context interface for sharing Data Source values between scripts and other tools.

Jython Example 1

```
from soaptest.api import *

def getKeywords(input, context):
   title = context.getValue("Books", "keywords")
    return SOAPUtil.getXMLFromString([title])
```

Jython Example 2

1.

```
def methodName():
  # code
```

2.
```
def methodName(input):
  # code
```

3.
```
def methodName(input, context):
  # code
```

In the above examples, "input" refers to either:

- The data passed from the output of the tool to which this Extension Tool is chained. For example, the Response SOAP Envelope of a SOAP Client OR
- The data that appears in the **Input** area of the Extension Tool. This is found at the bottom of the Extension Tool interface.

In the vast majority of cases, you can assume that the value passed to the "input" variable will be a primitive Jython string.

Generally, the "context" variable is determined dynamically by SOAtest and will be an instance of the Context class, found in the Extensibility API. In the case of an Extension Tool, the context will be an instance of ExtensionToolContext for this specific tool (as compared to other Extension Tools in your test suite).

JavaScript Example

```
var SOAPUtil = Packages.soaptest.api.SOAPUtil

function getKeywords(input, context) {
   title = context.getValue("Books", "keywords")
   return SOAPUtil.getXMLFromString([title])
}
```

Java Example

```
package examples;

import soaptest.api.*;
import com.parasoft.api.*;

public class Keyword {

    public Object getKeywords(Object input, ExtensionToolContext context)
            throws com.parasoft.data.DataSourceException {
        String[] titles = new String[1];
        titles[0] = context.getValue("Books", "keywords");
        return SOAPUtil.getXMLFromString(titles);
   }
}
```


Note

Java code such as this example must be compiled outside of SOAtest. You will need to make sure that you have the SOAtest jar files on the Classpath for your Java compiler in order to access packages from the SOAtest Extensibility API, in particular`webking.jar` and `com.parasoft.api.jar`.

For this Keyword example, the source code and compiled class file is available at SOAtestInstallRoot/build/examples. The SOAtestInstallRoot/build folder is already on the Classpath for SOAtest so you can use the Keyword example in a test suite. You could also put your own Java class files here to use the Java class in SOAtest. An alternative to using the SOAtestInstallRoot/build folder is to add the class file using the System Properties tab in the Preferences.

Reading File Contents into a String

To read the contents of a file into a String using a Jython script, you can use the following method:

```
from java.lang import *
from java.io import *
from soaptest.api import *

def getContents(input, context):
  contents = StringBuffer()
  reader = BufferedReader(FileReader(File("c:\Documents and Settings\jhendrick\Desktop\test.txt")))
  line = String()
  while line != None:
    line = reader.readLine()
    if line != None:
       contents.append(line)
    contents.append(System.getProperty("line.separator"))
reader.close()
return contents.toString()
```

Defining a Custom Assertion

Here is a sample Java script that can be used as a custom assertion in the XML Assertor tool. This script gets a value from a data source, prints that value along with the response value, and returns a boolean value based on whether or not the response value contains the data source value.

```
package examples;

import com.parasoft.api.*;

public class Comparison {

//Compares a value from a database data source to a value returned in a SOAP response
public boolean compareToDatabase(Object input, ScriptingContext context)
throws com.parasoft.data.DataSourceException {

//Gets values from database data source named "Books"
String value = context.getValue("Books", "Book");

//Prints database data source values along with SOAP response values in a message to the console
Application.showMessage("Value from database is " + value + ".\nValue from SOAP response is " +
input.toString() + ".");

//Verifies that the SOAP response value contains the database data source value
return input.toString().contains(value);
}
}
```

String Operations Using Extracted Values

Here is an example of a Jython script that extracts values from various sources (data bank, test suite variable, and environment variable) for use with string operations such as concatenation using the '+' operator...

```
from soaptest.api import *
from com.parasoft.api import *

# Gets a value from a data bank (line 10) or test suite variable (line 11),
# appends an environment variable value to the front of that value,
# and returns the modified value. Be sure to comment out line 11
# when using line 10 and vice versa.

def getModifiedString(context):
#value = context.getValue("Generated Data Source", "columnName")
value = context.getValue("testSuiteVariableName")

environmentVariableValue = context.getEnvironmentVariableValue("environmentVariableName")

modifiedString = environmentVariableValue + value

return modifiedString
```

This example is for a fairly specific use case but nonetheless uses some common methods from the SOAtest extensibility API.

Accessing a Window or Frame’s Document

These examples are intended for use when chaining an Extension tool to a Browser Playback Tool, used in a Web Scenario.

getDocument is overloaded as follows:

```
getDocument(); // gets the document for the main window
getDocument(String windowName); // gets the document for the window with the specified window name
getDocument(String windowName, String frameName); // gets the document for a frame within a
window
```

For example, the following code gets the content of a frame titled "mainPane" in the main window. It then uses the document to get the title text for the selected frame.

```
var Application = Packages.com.parasoft.api.Application;
var WebBrowserUtil = Packages.webking.api.browser2.WebBrowserUtil;

function getFrameDocument(input, context) {
   var document = input.getDocument("", "mainPane");
   var titles = document.getElementsByTagName("title");
   var title = titles.item(0);
   var value = WebBrowserUtil.getTrimmedText(title);
   Application.showMessage("title: " + value);
   return value;
}
```

Language-Specific Tips

Java

- When you specify a Java class in any of the scripting **Class** fields, you must specify a compiled class that is on your classpath. You can click the **Modify Classpath** link then specify it in the displayed Preferences page.
- If the class you are using is part of a package, you need to specify the complete package name as well as the class name (for example, `java.lang.String`)
- If one of your scripts uses a class that has been changed and recompiled within SOAtest or Virtualize, SOAtest or Virtualize will reload the class and use most recent class for object construction when you invoke the method. SOAtest and Virtualize do not function this way for any class whose package name starts with one of the following prefixes:
    - sun.
    - com.sun.
    - org.omg.
    - javax.
    - sunw.
    - java.
    - com.parasoft.
    - webtool.
    - wizard.
- To manually prompt SOAtest or Virtualize to reload a class that has been modified and recompiled, click **Reload Class**.

JavaScript

- The "legacy" SOAtest or Virtualize JavaScript emulation is based on FESI. We recommend using an alternative scripting engine such as Oracle Nashorn. For details, see:
    - Oracle Nashorn JavaScript
    - Other JSR 223 Scripting Languages
- You can call Java classes and methods that are on your classpath from inside JavaScript methods, or JavaScript tools.
    - For example, if you're using Nashorn and you want to call Application.report() (from the SOAtest or Virtualize Extensibility API) from inside JavaScript, you need to need to reference it as `Java.type("com.parasoft.api.Application"): var Application = Java.type("com.parasoft.api.Application") Application.report("Message", "Result Window")`
    - If you decide to use the legacy engine (not recommended) and you want to call Application.report() (from the SOAtest or Virtualize Extensibility API) from inside JavaScript, you need to need to reference it as `Packages.com.parasoft.api.Application.report()`. You could also reference it by prepending the name with `Packages` and the name of the package where the Java class lives as follows `var Application = Packages.com.parasoft.api.Application Application.report("Message", "Result Window")`
- To check that the specified script is legal and runnable (or to add method entries to the **Method** box), right-click the **File** or **Text** text field (click whichever one you used to specify your script), then choose **Evaluate** from the shortcut menu.

Jython

For details on Jython (an implementation of Jython that is integrated with Java), including information on how to write Jython and also how to invoke Java classes from inside Jython, visit http://www.jython.org. Note that SOAtest and Virtualize ship with Jython 2.5.2.

- If you are using Jython scripts, you might need to specify your jython.home and jython.path variables in the Scripting tab of the Preferences panel. Both variables are used to locate Jython modules, and Jython code that does not import any Jython modules can use the Jython scripting support without setting either variable. jython.home specifies the Jython installation directory. jython.path is used to add to your path modules that are not in your jython.home/Lib directory. Multiple paths can be listed in jython.path, while jython.home must be a single directory. If you set the jython.home and jython.path variables, you need to restart SOAtest or Virtualize before the changes will take effect.
    - If you decide to import optional modules and you do not have Jython 2.5.2 installed, you will need to download and install Jython 2.5.2 from jython.org. For Linux, use java -jar jython_installer-2.5.2.jar. For Windows, double-click jython_installer-2.5.2.jar.

- To check that the specified script is legal and runnable (or to add method entries to the **Method** box), right-click the **File** or **Text** text field (click whichever one you used to specify your script), then choose **Evaluate** from the shortcut menu.

Groovy

Here is a sample Groovy script:

```
import com.parasoft.api.*;

boolean customAssertion(Object input, ScriptingContext context) {
    String value = context.getValue("Books", "title");
    Application.showMessage("Value from data source is " + value)
    Application.showMessage("Value from SOAP response is " + input.toString())
    return input.toString().contains(value);
}
```

JSR 223 Scripting Languages

You can configure SOAtest and Virtualize to recognize any scripting engines that implement the JSR 223 "Scripting for the Java Platform" specification. Oracle Nashorn is available by default; others can be added.

Oracle Nashorn JavaScript

The Oracle Nashorn ECMAScript engine is available by default because it is included with Java. Here is a sample Oracle Nashorn script:

```
var Application = Java.type("com.parasoft.api.Application")

function customAssertion(input, context) {
    value = context.getValue("Books", "title");
    Application.showMessage("Value from data source is " + value)
    Application.showMessage("Value from SOAP response is " + input.toString())
    return input.toString().contains(value);
}
```

Mozilla Rhino JavaScript

Mozilla Rhino is the original ECMAScript engine included with the Java runtime. It is the predecessor to Oracle Nashorn, which has replaced Mozilla Rhino as of Java 8. SOAtest and Virtualize versions prior to 9.10 shipped with Java 7, which included Mozilla Rhino. Scripts saved in earlier versions of SOAtest and Virtualizewith the Mozilla Rhino engine will now run in the new Oracle Nashorn with Mozilla compatibility extensions loaded. The compatibility extensions are provided by Oracle as part of Nashorn and are intended for such purposes. By loading the compatibility extensions, scripts will continue to function as before, but with significantly improved runtime performance.

Scripts originally saved using Mozilla Rhino will now show **JavaScript (Mozilla Rhino compatible)** in the **Language** selection box, indicating that the script will run with Nashorn but with the Mozilla compatibility extensions. If you create a new script, the **JavaScript (Mozilla Rhino compatible)** option will not be available; you will see only **JavaScript (Oracle Nashorn)**, which does not load the Mozilla compatibility extensions. If you need to load the compatibility extensions in this case, simply `add load("nashorn:mozilla_compat.js");` to the top of your script. For more detail, see https://wiki.openjdk.java.net/display/Nashorn/Rhino+Migration+Guide#RhinoMigrationGuide-Compatibility-script.

Since Oracle Nashorn is the replacement for Mozilla Rhino, use of Mozilla Rhino is not recommended. However, Mozilla Rhino can be made available to SOAtest and Virtualize in the same manner as any other JSR 223 compatible script engine (see Other JSR 223 Scripting Languages). Mozilla Rhino includes two jars which must be added to the classpath in the Preferences panel (under System Properties): Rhino js.jar and js-engine.jar. For details, see https://wiki.openjdk.java.net/display/Nashorn/Using+Rhino+JSR-223+engine+with+JDK8.

Pre-built jars can also be found on maven central:

- http://search.maven.org/#search%7Cga%7C1%7Cg%3A%22org.mozilla%22%20a%3A%22rhino%22
- http://search.maven.org/#search%7Cga%7C1%7Ca%3A%22rhino-js-engine%22

Other JSR 223 Scripting Languages

To make another JSR 223 scripting language available, add the appropriate engine's scripting engine to the classpath (go to the Parasoft Preferences panel and add it under the **System Properties** area).

For example, you could add the JRuby scripting engine, which implements JSR 223. To do this, you would download JRuby from http://jruby.org/ then add jruby.jar to the classpath.

Here is a sample JRuby script:

```
require 'java'
Application = com.parasoft.api.Application

def doSomething(input, context)
    Application.showMessage("hello world")
end
```

Additional Extensibility Resources

For details on how to create and apply an Extension tool that executes a custom script you have written, see Extension Tool for Custom Scripting.