Extension Tool for Custom Scripting

This topic explains how to create and apply an Extension tool. The tool performs custom operations written in Java, JavaScript, Oracle Nashorn, Groovy, Jython, or other scripting engines that implement the JSR 223 "Scripting for the Java Platform" specification.

Sections include:

Understanding the Scripting/Extension Tool

The Extension tool allows you to implement a custom operation independently or in conjunction with your existing

test suite.

This additional functionality gives you the ability to customize

SOAtest

to your specific needs and is limited only by the capabilities of the scripting language that you are working with.

The scripts may accept zero, one, or two arguments. The source of the output can be a derived from a client request, a server response, the output of another tool, an element of a rule, or it can be user defined beforehand. This is useful if you want to perform application-specific checks that cannot be expressed with a rule (for example, if you want to check if a given output matches a record in a database). Or you can design the script to perform any specific function that would be helpful to you.

If your method accepts zero arguments, then it does not matter what file is selected when it is run. If the method accepts one argument then the input will be taken from the file selected or the tool to which it is attached. If the method accepts two arguments then the first will be taken from the file or preceding tool and the second will take contextual information about the file. For more information o n the types of context arguments that can be applied, see the Scripting API by choosing **Parasoft> Help> Scripting API**.

You can also create scripts that execute each time you start

SOAtest.

To do this, create a Jython or JavaScript script, then add it to the following

location

```
<SOAtest_Installation_Directory>/plugins/com.parasoft.xtest.libs.web_<soatest_version>/root/startup
```

Creating a Custom Script/Extension Tool

The following procedure describes how to add an Extension tool to the toolbar. You can also create custom methods in a test suite (in SOAtest) or Responder suite (in Virtualize) by adding an Extension tool, then completing the same parameters in the Extension parameters panel.

To add a custom script/Extension tool to SOAtest or Virtualize:

1. Add an Extension toolusing one of the techniques described in End-to-End Test Scenarios.
2. Double-click the Extension tool's

   Test Case Explorer

   node. The tool configuration panel will open on the right.
3. Give your method a name in the **Name** field.
4. If a return value for this tool indicates the tool's success, select the **Exit code indicates success** check box. If this check box is not selected, the return value of the method is ignored regardless of whether the tool succeeded or failed.
5. From the **Language** box, select the language that your method is or will be written in.
6. 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 in the UI.

- To use an existing file, select the **File** radio button and click **Browse.** Select the file from the file chooser that opens, then click **OK** to complete the selection.
- To create the method in the UI from scratch, 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. Any problems found will be reported.

7. Click **Evaluate** to check that the script will run (does not contain syntax errors).
8. In the **Error message** field, specify what error message should be reported if the tool fails.
9. Select the appropriate argument from the **Method** box at the bottom of the panel. This list will be composed of any definitions contained in your script. Since a script can contain multiple arguments, you can select the one that you want to use in this method.

Using Variables in Your Script?

If your script returns content that contains ${} format variables such as `${value}`, those variables will be resolved before any attached output tools process the content. If you do not want those variables to be resolved, be sure to escape the variables in your script.

For example, assume that your original script has:

```
def handler(input, context) {
            def output = '{ "field": "${value}" }';
            com.parasoft.api.Application.showMessage(output);
                return output;
}
```

To prevent `${value}` from being processed, you would modify it to:

```
def handler(input, context) {
    def output = '{ "field": "${value}" }';
    output = output.replace('${', '\\${');
    com.parasoft.api.Application.showMessage(output);
        return output
}
```

Additional Scripting Resources

For an overview of issues related to the scripting feature and its various applications, see Extensibility and Scripting Basics.For a step-by-step demonstration of how to apply custom scripting in SOAtest, see Extending SOAtest with Scripting.