

DevTest 8.0 – Custom Extensions

Summary

DevTest 8.0 supports customization of its components by scriptable or programmable extensions. These extensions are envisioned as configuration steps to complement the rich set of features available already out of the box. Custom DevTest Extensions might be necessary to implement accommodating customer specific configuration of the system under test, to match or to verify correctness of complex dependencies and data integrity, for instance.

This document intends to help engineers, who are new to the concept of scripted extensions in DevTest solutions, to get started. It is not supposed to replace current product documentation, but to complement it: information about scripted extensions is scattered to various locations in product documentation, located close to the DevTest components they can extend. This document pulls together all this information into this single document. Because of its focus on scripting, commonalities, specifics, recommendations, and finally best practices of the multiple scripting environments are described and explained.

This document does not cover introductions into scripting languages used in DevTest.

Some extension capabilities become available with Service Packs for DevTest 8.0 only. This is noted along with the description.

This document covers DevTest 8.0.2.

Document History

Date	Version	Author	Comment
06/02/2015	1.0	Cameron Bromley Rick Brown Ulrich Vogt	Initial version. Covers DevTest 8.0.1.
04/10/2015	1.1	Ulrich Vogt	8.0.2 enhancements <ul style="list-style-type: none">• Coloring custom events• Scripted data• Document references updated to wiki for 8.0.2

Contents

Summary	1
Document History	1
Legal	4
Copyright notice.....	4
Target Audience	4
Documentation	5
Extension Capabilities	5
Extension by Scripting.....	5
Default Scripting Engine.....	6
Groovy.....	6
JavaScript	6
BeanShell.....	6
Velocity	7
Scripting support in DevTest 8.0.....	7
Java Script Step	7
Common.....	7
Injected Variables and Properties.....	7
TestExec class	8
Scripts in Mobile Testing.....	17
VSE Classes	17
Security	19
Logging	19
Events.....	21
Sharing Data	23
Debugging	26
Editor.....	27
Configuration Area.....	27
Object Selector.....	28
Code Editor	29
Status area	30
Scripted Expression.....	31
Sample.....	31
Test Step - Execute Script (JSR-223).....	33

DevTest Product Documentation.....	33
Input Parameters	33
Output Parameters	33
Logging output	34
Editor.....	34
Sample.....	35
Scripted Assertion	37
DevTest Product Documentation.....	37
Input Parameters	37
Output Parameters	37
Logging output	37
Editor.....	37
Sample.....	38
Virtual Service Router Step	39
DevTest Product Documentation.....	39
Input Parameters	39
Output Parameters	39
Logging output	39
Editor.....	40
Sample.....	40
Match Script in Virtual Service Images	42
DevTest Product Documentation.....	42
Input Parameters	42
Output Parameters	42
Logging Output.....	42
Match Script Editor	43
Sample 1.....	44
Sample 2.....	44
Scriptable Data Protocol	46
DevTest Product Documentation.....	46
Scripts.....	46
Input Parameters	46
Output Parameters	46
Logging output	47
Editor.....	47

Best Practices	47
Errata.....	50
Sample 1.....	50
Sample 2.....	53
Scripted Dataset.....	54
DevTest Documentation	54
Input Parameters	54
Output Parameters	54
Editor.....	54
Sample.....	55
Appendix	56
Performance considerations.....	56
Sample Code	56

Legal

- What kind of CA support is provided with a custom extension? Custom extensions are not covered by common CA product support conditions. Special support can be negotiated. Please contact your CA representative.
- What happens when a new version of DevTest is installed? A custom extension might need to be recompiled to meet the differences in DevTest API. CA strives for DevTest API backward compatibility.
- What happens if the customer finds a problem with the custom extension? If no special support contract is in place CA Professional Services can be contracted to find the problem, fix it and deploy it to every place it has been installed (all DevTest workstations, all DevTest servers).

All sample code provided in this document

Copyright notice

Copyright © 2015 CA, Inc. All rights reserved. All marks used herein may belong to their respective companies. This document does not contain any warranties and is provided for informational purposes only. Any functionality descriptions may be unique to the customers depicted herein and actual product performance may vary.

Target Audience

This document is intended for customers, partners, and CA field personnel familiar with DevTest, who want to create scripts to extend or to customize existing functionality in DevTest solutions.

Programming skills are required, as well as basic knowledge of scripting languages such as JavaScript, Groovy or BeanShell. Java knowledge is required to understand java based code samples.

It is strongly recommended to review related product documentation. References to related DevTest product guide sections on CA support are given.

Documentation

- [1] DevTest Solutions: Using Service Virtualization – DevTest 8.0.2 Product Documentation [Using CA Service Virtualization](#)
- [2] DevTest Solutions: Using Agents – DevTest 8.0.2 Product Documentation [Using CA Application Test](#)
- [3] DevTest Solutions: Using the SDK – DevTest 8.0.2 Product Documentation [Using the SDK](#)
- [4] DevTest Solutions: Administering – DevTest 8.0.2 Product Documentation [Administering](#)
- [5] SDK JavaDoc – `{{LISA_HOME}}\doc\SDKJavaDoc\index.html`

Extension Capabilities

DevTest 8.0 allows for custom extensions in various components

- Test cases
 - Custom Java Test step - Java
 - Custom JSR-223 script Test step - Script
 - Custom Assertions to verify customer specific dependencies [3] – Script/Java
 - Custom Filters to store additional data in properties [3] – Java
 - Custom Companion [3] – Java
- Staging documents
 - Custom reports to report on customer specific events [3] – Java
 - Custom Report Metrics to extract customer specific metrics from test cases [3] – Java
- Agent
 - Modify agent behavior [2] – Java
 - Manipulate data captured by the agent [2] – Java
 - Execute additional steps during recording or playback before or after virtualized methods are called [2] – Java
- Broker
 - Change data in frames [2] – Java
 - Add/remove frames [2] – Java
 - Customize the stitching algorithm for data from different agents [2]
- Virtual Services
 - Match script in Virtual service images [1] – Script
 - Virtual Service Router step in Virtual Service models
 - Scriptable Data Protocols to modify service requests or responses while recording [1] – Script
 - Custom Data Protocol Handlers modify service requests or responses while recording [1] – Java

Extension by Scripting

Out of the box DevTest 8.0 supports scripting languages

- BeanShell

- Groovy
- JavaScript
- Velocity

BeanShell was the only scripting language available and supported in previous product versions. It is now being deprecated, because it is rather slow if compared to JavaScript and Groovy, and because it is not maintained actively by the internet community any more. Therefore BeanShell is discouraged to use for new scripts. BeanShell scripting remains supported for backward compatibility.

BeanShell scripts are processed by a BeanShell interpreter. When possible, JavaScript and Groovy scripts are compiled into Java byte code first before run. Therefore JavaScript and Groovy scripts have better performance once compiled than BeanShell scripts.

Default Scripting Engine

BeanShell is the default scripting language in DevTest 8.0. To change the default scripting language to Groovy, for instance, edit file 'local.properties' and add following line:

```
##=====DevTest Default Scripting Engine=====
lisa.scripting.default.language=groovy
```

DevTest can be configured to support additional scripting engines. Please see [Enabling Additional Scripting Languages](#) for details.

The scripting languages that are installed and found by the runtime environment are listed out in the relevant log file (e.g. workstation.log) once a script is called the first time:

```
2015-01-22 09:50:46,346Z (10:50) INFO com.itko.lisa.test.UserScriptNode -
JSR223 language engine Groovy Scripting Engine 2.0 for language Groovy 2.3.3
(groovy Groovy) from file:/D:/DevTest-800GA/lib/shared/groovy-all-2.3.3.jar

2015-01-22 09:50:46,346Z (10:50) INFO com.itko.lisa.test.UserScriptNode -
JSR223 language engine BeanShell Engine 2.1.8 for language BeanShell 2.1.8
(beanshell bsh) from file:/D:/DevTest-800GA/lib/shared/bsh-2.1.8.jar

2015-01-22 09:50:46,347Z (10:50) INFO com.itko.lisa.test.UserScriptNode -
JSR223 language engine Mozilla Rhino 1.7 release 3 PRERELEASE for language
ECMAScript 1.8 (js rhino JavaScript javascript ECMAScript ecmaScript) from
<unknown>

2015-01-22 09:50:46,347Z (10:50) INFO com.itko.lisa.test.UserScriptNode -
JSR223 language engine velocity 1.5 for language velocity 1.5 (Velocity
velocity) from file:/D:/DevTest-800GA/lib/shared/itko-velocity-engine-1.7.jar
```

Each log entry lists the valid names for the scripting engine. These names can later be used as language specifiers to determine the scripting engine to use.

Groovy

Introduction, documentation, tutorials on groovy can be found on <http://groovy.codehaus.org/>.

JavaScript

Introduction, documentation, tutorials and samples are found on JavaScript at <http://www.w3schools.com/js/>.

BeanShell

BeanShell is run by a Java interpreter. In a BeanShell script you can type normal Java statements and expressions and display the results.

Introduction, documentation and samples on BeanShell can be found at <http://www.beanshell.org/>. DevTest Product documentation includes a basic introduction into BeanShell at [Using BeanShell in DevTest](#).

Velocity

Velocity can be downloaded from <https://velocity.apache.org/index.html>. Documentation on Velocity is available at <https://velocity.apache.org/engine/releases/velocity-1.5/user-guide.html>.

Scripting support in DevTest 8.0

Extending the existing rich set of DevTest functionality by scripting is available at following places:

- **Scripted Expressions** - Anywhere you use `{{expressions}}` you can specify a scripting language
- **Execute Script (JSR-223)** - A test case or virtual service model can be extended by a test step that executes a script. This script can run additional logic to execute commands and return data to the test case context for other tests steps to use. This test step replaces the 'Java Script Step' available in previous product releases, which is deprecated in DevTest 8.0.
- **Scripted Assertion** – An assertion can be added to a test step that executes a script in order to assure and verify test step results
- **Match Script** - For virtual services images complex algorithms can be implemented in VSE by a script to find a matching transaction to a client request.
- **Scriptable Data Protocol Handler (DPH)** – If built-in data protocol handlers do not correctly translate client data into a format VSE understands or vice versa, a scriptable DPH can be developed to bridge the gap.
- **Virtual Service Router Step** – This step routes a request from a virtual service listen step to the response selector step and the protocol-specific live invocation step, or both. The decision is made based on the current execution mode for the running model. If running in DYNAMIC mode a script can be used to determine the route of the request.

Java Script Step

The 'Java Script Step', which is available previous product releases, but deprecated in DevTest 8.0, will not be covered. The 'Java Script Step' step is functional equivalent to 'Execute Script (JSR-223)' step configured for BeanShell scripting language with a property scope of 'Test state and system properties'.

Common

This section covers information that is common to all the scriptable extensions in DevTest. Deviations will be covered in the sections covering specifics of extensions.

Injected Variables and Properties

DevTest gives scripts access to the runtime environment by different means. It supplies built-in – injected – variables to access information available in the current test case or virtual service.

- 'textExec' object – specifies the current test execution environment. The testExec object includes the state and supporting behaviors for running the test.
- 'lisa_vse_request', 'lisa_vse_response' objects – specifies the current live transaction request, current live or recorded transaction response in a scriptable Data Protocol Handler (DPH) of a virtual service in action.

- ‘incomingRequest’ and ‘sourceRequest’ objects – specify the live incoming and the recorded source transaction request in Match Scripts
- ‘_logger’ object – specifies a logger object available to the current script environment
- Test state properties (optional) – provide shortcut access to test properties. This is also available by ‘testExec’ via method calls
- System properties (optional) – provides shortcut access to java system properties, also available via ‘testExec’ method calls
- ‘_webDriver’ – references the MobileSession object that is associated with the test case, and is used to script Appium. ‘_webDriver’ works during runtime only. ‘_webDriver’ is available starting with DevTest 8.0.1.

When creating a script in DevTest the developer can often choose the scope of variables to be injected into the script:

- **‘testExec’ and ‘_logger’ only:** only the ‘testExec’ and ‘_logger’ objects are available to the script (recommended)
- **Test state properties:** direct access to properties that provide information about the test case. ‘testExec’ and ‘_logger’ objects are also available
- **Test state and system properties:** all properties for the test case and system. This is the same scope as in previous product versions. ‘testExec’ and ‘_logger’ objects are also available.

The caveat to the wide scope of accessibility to the test environment is the fact that the broader the scope is the longer it takes to setup the scope for the script and to start the script.

Test state and system properties are also available via ‘testExec’ object.

The syntax to access the variables may vary on language.

Some scripting entry points have other variables, e.g. Data Protocol Handlers and Match Scripts have access to more variables, see specific examples below.

{{some_property}}

Referring to a property in property tags *{{some_property}}* in a script is supported. The property is substituted for the property value at runtime before the script is executed. This use of property tags is discouraged, though, as parsing for replacements slows down execution of the script setup.

When retrieving a property from test case that includes property tags there are methods, which replace properties by their values.

TestExec class

The TestExec class supplies several methods. The following list is just an overview of methods available for this class. Please see [5] for more details. Of interest are mostly how to retrieve and to set properties of the current test case under execution:

java.lang.Object getStateValue(String strKey)	Returns the java.lang.Object of the given key
boolean getStateBoolean(String strKey, boolean bDefault)	Returns the value of <i>strKey</i> as a boolean value. If property <i>strKey</i> does not exist, has no value or no boolean value, bDefault is returned
int getStateInt(String strKey, int nDefault)	Returns the value of <i>strKey</i> as an int value. If property <i>strKey</i> does not exist, has no value or no int value, nDefault is returned.

String getStateString(String strKey, String strDefault)	Returns the value of <i>strKey</i> as a String value. If property <i>strKey</i> does not exist, has no value or no String value, <i>strDefault</i> is returned.
void setStateValue(String strKey, java.lang.Object value)	Sets property <i>strKey</i> to value of type java.lang.Object
void log (String shortMsg, String longMsg)	Pushes a 'Log message' event message
void warn (String shortMsg, String longMsg)	Pushes a 'Step warning' event message
void raiseEvent(int eventID, String shortMsg, String longMsg)	Send a custom event message

For samples on retrieval or setting of properties see [Reading a Property](#) and [Setting a Property](#). Samples for different scripting languages are given in [Script Samples Accessing testExec objects and Properties](#). Samples to create event messages are described in [Events](#).

There are many other methods available through TestExec objects - see [5] for details.

Reading a Property

Almost every script needs to retrieve data from test case. The recommended way to do that is using a 'testExec.getState*()' method.

The '*testExec.getStateValue("Key")*' method returns the value of property 'Key' as Object. If Key is not set the returned string is empty, if 'Key' does not exist the returned value is 'null'.

Then there are type specific 'testExec.getState*()' methods, such as '*testExec.getStateString("Key", defaultString)*', '*testExec.getStateInt("Key", defaultInteger)*', '*testExec.getStateBoolean("Key", defaultBoolean)*'. These type specific retrieval methods return their respective default value in case property 'Key' is undefined. For Boolean and Integer values the default string is also returned if property 'Key' is not set. For String values the empty string is returned if property 'Key' exist but is not set.

Sample

For each of the String, Boolean, and Integer type two properties are defined, one set with a valid value, another one not set.

Available Objects		
Name	Type	Value
AA_Boolean1	java.lang.Boolean	true
AA_Boolean2	java.lang.String	
AA_Integer1	java.lang.Integer	1234
AA_Integer2	java.lang.String	
AA_String1	java.lang.String	This is a test string
AA_String2	java.lang.String	

The following sample groovy code executes for each property '*testExec.getStateValue("Key")*' and the type specific method to retrieve the property value. Additional both methods are applied to retrieve an undefined property.

String

```

Script
1 strInfo1 = testExec.getStateValue("AA_String1")
2 strInfo2 = testExec.getStateValue("AA_String2")
3 strInfo3 = testExec.getStateValue("AA_String3")
4 _logger.info("1. Info-1="+strInfo1+" - Info-2="+strInfo2+" - Info-3="+strInfo3+".")
5
6 strInfo1 = testExec.getStateString("AA_String1","Default String")
7 strInfo2 = testExec.getStateString("AA_String2","Default String")
8 strInfo3 = testExec.getStateString("AA_String3","Default String")
9 _logger.info("2. Info-1="+strInfo1+" - Info-2="+strInfo2+" - Info-3="+strInfo3+".")
10

```

Please note the differences in the output when retrieving the value for the undefined property 'AA_String3' in 'Info-3' output. Retrieving an empty property returns identical values ('Info-2').

```

INFO com.itko.lisa.script.logger - 1. Info-1=(This is a test string) - Info-2=() - Info-3=(null).
INFO com.itko.lisa.script.logger - 2. Info-1=(This is a test string) - Info-2=() - Info-3=(Default String).

```

Boolean

```

11 bInfo1 = testExec.getStateValue("AA_Boolean1")
12 bInfo2 = testExec.getStateValue("AA_Boolean2")
13 bInfo3 = testExec.getStateValue("AA_Boolean3")
14 _logger.info("3. Info-1="+bInfo1+" - Info-2="+bInfo2+" - Info-3="+bInfo3+".")
15
16 bInfo1 = testExec.getStateBoolean("AA_Boolean1", false)
17 bInfo2 = testExec.getStateBoolean("AA_Boolean2", false)
18 bInfo3 = testExec.getStateBoolean("AA_Boolean3", false)
19 _logger.info("4. Info-1="+bInfo1+" - Info-2="+bInfo2+" - Info-3="+bInfo3+".")
20

```

Please note the differences in the output when retrieving the value of the empty property 'AA_Boolean2' in 'Info-2' output and of the undefined property 'AA_Boolean3' in 'Info-3' output. For Boolean and Integer type the default value is returned if respective property is not set or does not exist.

```

INFO com.itko.lisa.script.logger - 3. Info-1=(true) - Info-2=() - Info-3=(null).
INFO com.itko.lisa.script.logger - 4. Info-1=(true) - Info-2=(false) - Info-3=(false).

```

Integer

```

21 nInfo1 = testExec.getStateValue("AA_Integer1")
22 nInfo2 = testExec.getStateValue("AA_Integer2")
23 nInfo3 = testExec.getStateValue("AA_Integer3")
24 _logger.info("5. Info-1="+nInfo1+" - Info-2="+nInfo2+" - Info-3="+nInfo3+".")
25
26 nInfo1 = testExec.getStateInt("AA_Integer1", 0)
27 nInfo2 = testExec.getStateInt("AA_Integer2", 0)
28 nInfo3 = testExec.getStateInt("AA_Integer3", 0)
29 _logger.info("6. Info-1="+nInfo1+" - Info-2="+nInfo2+" - Info-3="+nInfo3+".")

```

Please note the differences in the output when retrieving the value of the empty property 'AA_Integer2' in 'Info-2' output and of the undefined property 'AA_integer3' in 'Info-3' output. For Boolean and Integer type the default value is returned if respective property is not set or does not exist.

```

INFO com.itko.lisa.script.logger - 5. Info-1=(1234) - Info-2=() - Info-3=(null).
INFO com.itko.lisa.script.logger - 6. Info-1=(1234) - Info-2=(0) - Info-3=(0).

```

Using `parseInState()`

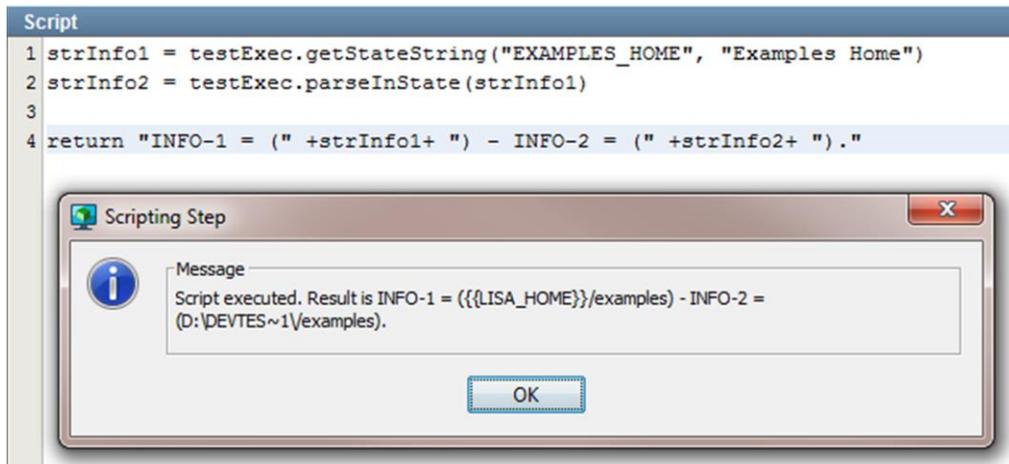
Retrieving a property that includes property tags, i.e. `{{some_prop}}`, requires the use of method `'testExec.parseInState()'` in order to replace a property by its current value.

Sample

Using property 'EXAMPLES_HOME'

Available Objects		
Name	Type	Value
EXAMPLES_HOME	java.lang.String	{{LISA_HOME}}/examples

The following groovy script returns the real path only after applying the 'parseInState()' method in variable 'strInfo2'



```

Script
1 strInfo1 = testExec.getStateString("EXAMPLES_HOME", "Examples Home")
2 strInfo2 = testExec.parseInState(strInfo1)
3
4 return "INFO-1 = (" +strInfo1+ ") - INFO-2 = (" +strInfo2+ ")."

```

Scripting Step

Message

Script executed. Result is INFO-1 = {{LISA_HOME}}/examples - INFO-2 = (D:\DEVTEST~1\examples).

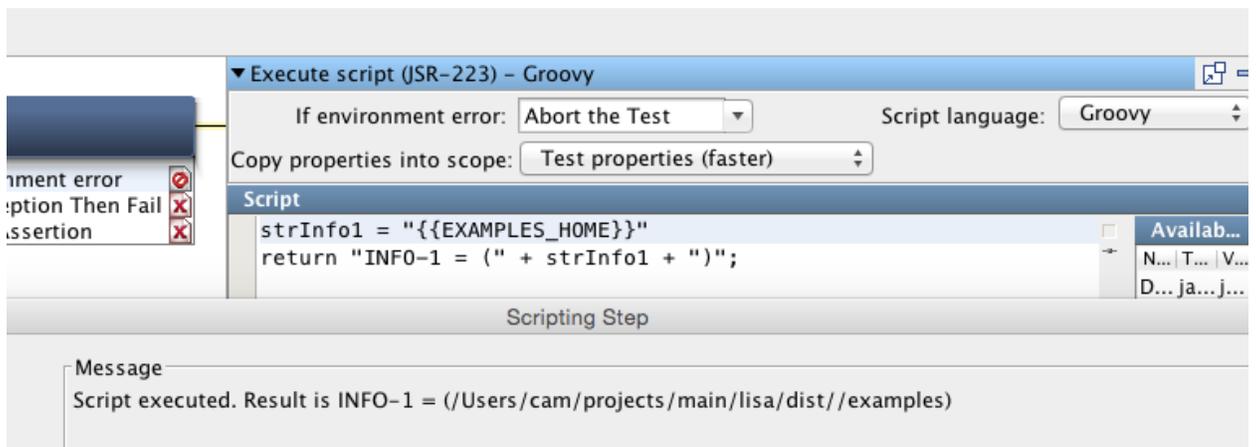
OK

Note that scripts are parsed for {{properties}} before they are executed, so the following script is equivalent to the above:

```

strInfo1 = "{{EXAMPLES_HOME}}"
return "INFO-1 = (" + strInfo1 + ")";

```



Execute script (JSR-223) - Groovy

If environment error: Abort the Test Script language: Groovy

Copy properties into scope: Test properties (faster)

Script

```

strInfo1 = "{{EXAMPLES_HOME}}"
return "INFO-1 = (" + strInfo1 + ")";

```

Scripting Step

Message

Script executed. Result is INFO-1 = (/Users/cam/projects/main/lisa/dist//examples)

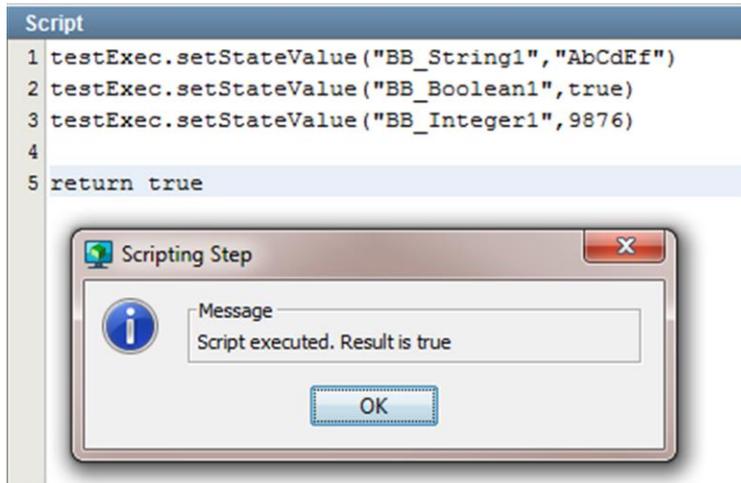
Setting a Property

Scripts return results when finished. Depending on the scripting language there are different means to do that, usually by a return value.

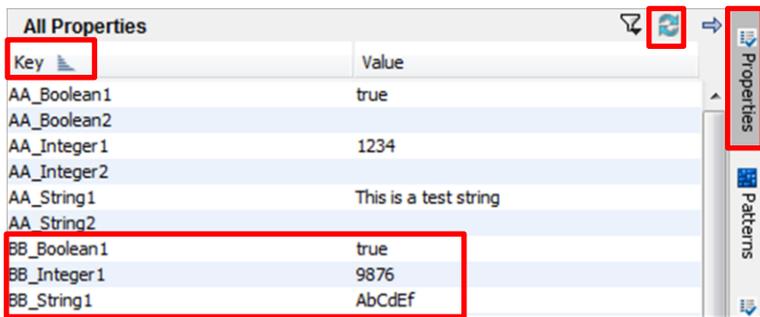
If there is a need to return more than one value a script can create and update properties of a test case. 'testExec.setStateValue("Key", value)' sets a property 'Key' to a string **value**.

Sample

This groovy code snippet (creates and) sets three new properties:



Opening the 'Properties pane', clicking on the 'Refresh' button and sorting the properties by 'Key' lists all the properties of the test case.

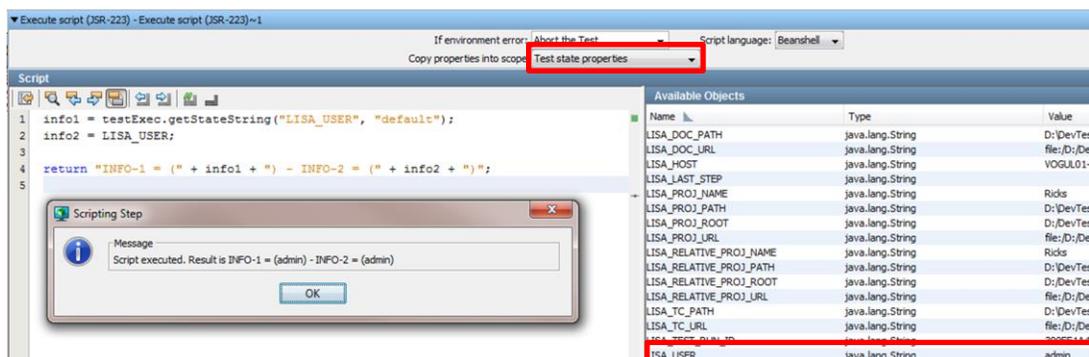


Access to Test state and system property

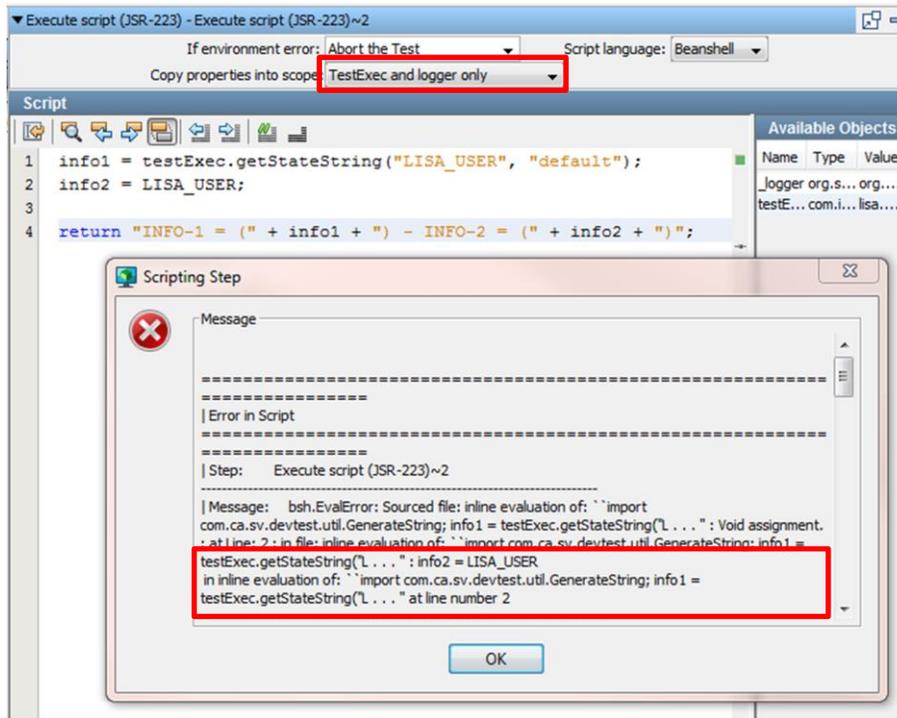
Test state and system properties can be accessed directly by name or via 'testExec'.

Sample

Please note the property scope, which includes direct access to test case properties, as used in line #2 of the code sample.



The same code will fail in line #2 if the scope is changed to 'TestExec and logger only', because property 'LISA_USER' will then be unavailable.



Access to Properties that not in Java variable format

When a script is being set up, system and test case properties will be copied into the runtime scope of the script as variables. These variables need to have valid JavaScript/BeanShell/Groovy variable names. Variable names with dots in them are not valid identifiers. So a property 'foo.bar' is converted into a form suitable for a variable name, in this case of we change all '.' chars to '_'. So we end up with a variable named 'foo_bar'. 'Available Objects' list contains already converted property names.

With 'testExec' unconverted, original property names have to be used. Please see the following sample.

Sample

Property scope is 'Test state and system properties' and there is a system property 'file.encoding' available. 'Available Objects' has this property already converted, ready to use as a variable in the script.

Available Objects		
Name	Type	Value
file_encoding	java.lang.String	UTF-8

The following sample script shows how to access this property by its variable name or by 'testExec'. 'testExec.getStateString()' would return the 'default' string to indicate that this property is unavailable or not set.

Reviewing the script output shows that

- Direct access to property 'file_encoding' works as expected (INFO-1).
- Reading the converted name by 'testExec' does not return the string (INFO-2).
- Reading the original system property name by 'testExec' works as described above.

The screenshot shows the 'Execute script (JSR-223)' window. The 'Copy properties into scope' dropdown is set to 'Test state and system properties'. The script contains the following code:

```

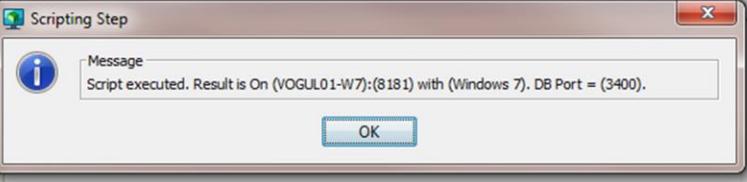
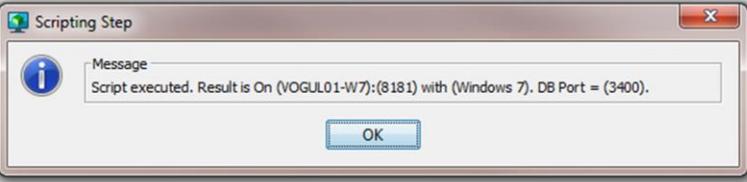
1 // Use of properties with
2 info1 = file_encoding;
3 info2 = testExec.getStateString("file_encoding", "default");
4 info3 = testExec.getStateString("file.encoding", "default");
5
6 return "INFO-1=(" + info1 + ") - INFO-2=(" + info2 + ") - INFO-3=(" + info3 + ")";
7
8

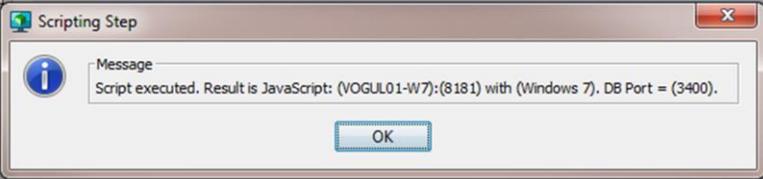
```

The message dialog displays the output: "Script executed. Result is INFO-1=(UTF-8) - INFO-2=(default) - INFO-3=(UTF-8)".

Script Samples Accessing testExec objects and Properties

Following are some samples of retrieving and setting properties in scripting languages BeanShell, JavaScript and Groovy.

<pre> 1 /* 2 This is similar to the deprecated scripting step in that 3 we are using beanshell. 4 */ 5 6 // this goes to the DevTest log file. Useful for debugging 7 _logger.info("Hello from BeanShell at " 8 + testExec.getStateObject("LISA_PROJ_NAME")); 9 10 osname = os_name; 11 host_name = testExec.getStateString("LISA_HOST", "No Host"); 12 port_no = testExec.getStateInt("PORT", 9989); 13 port_no = port_no + 101; 14 port_db = DBPORT; 15 port_db = port_db + 94; 16 17 testExec.setStateObject("SCRIPT_OUT", "BeanShell on " + host_name + 18 ":" + port_no + " with " + osname); 19 20 return "On (" + host_name + "):(" + port_no + ") with (" + osname + 21 "). DB Port = (" + port_db + ")."; 22 </pre> 	<p>BeanShell sample code:</p> <ul style="list-style-type: none"> • #7: Logs information to current logging file, incl. reading property 'LISA_PROJ_NAME' via TestExec object • #10: Sets var 'osname' via property 'os_name', which is in fact a system property 'os.name' • #11: Reads String property 'LISA_HOST' via TestExec object with default string 'No Host' • #12: Reads Integer property 'PORT' via TestExec object with default value of 9989 • #13: Adds to read port number to proof integer type • #14: Reads property DBPORT • #15: Adds to read DBPORT property to proof integer type. • #17: Sets property SCRIPT_OUT via TestExec object • #20: Returns some string.
<pre> 1 // built-in logger object 2 _logger.info("Hello from Groovy at " + LISA_PROJ_NAME) 3 4 // interact directly with testExec 5 testExec.setStateObject("Hello", "Groovy World") 6 7 osname = System.getProperty("os.name") 8 host_name = testExec.getStateString("LISA_HOST", "No Host") 9 port_no = testExec.getStateInt("PORT", 9989) 10 port_no = port_no + 101 11 port_db = DBPORT 12 port_db = port_db + 94 13 14 testExec.setStateObject("SCRIPT_OUT", "Groovy on " + host_name + 15 ":" + port_no + " with " + osname) 16 17 return "On (" + host_name + "):(" + port_no + ") with (" + 18 osname + "). DB Port = (" + port_db + ")." 19 20 </pre> 	<p>Groovy sample code:</p> <ul style="list-style-type: none"> • #1: Logs information to current logging file, incl. reading property 'LISA_PROJ_NAME' directly • #5: Sets property Hello via TestExec object • #7: Sets var 'osname' via 'System' call • #8: Reads String property 'LISA_HOST' via TestExec object • #9 Reads Integer property 'PORT' via TestExec object with default value of 9989 • #10: Adds to read port number to proof integer type • #11: Reads property DBPORT • #12 Adds to read DBPORT property to proof integer type. • #14: Sets property SCRIPT_OUT via TestExec object • #17: Returns some string.

<pre> 1 // built-in logger object 2 _logger.info("Hello from JavaScript at " + LISA_PROJ_NAME) 3 4 // interact directly with testExec 5 testExec.setStateObject("Hello", "JavaScript World"); 6 host_name = testExec.getStateString("LISA_HOST", "No Host") 7 osname = os_name; 8 port_no = testExec.getStateInt("PORT", 0) 9 port_no = port_no + 101 10 port_db = DBPORT 11 port_db = port_db + 94 12 13 testExec.setStateObject("SCRIPT_OUT", "JavaScript on " + host_name + 14 ":" + port_no + " with " + osname) 15 16 // implicit return value back 17 // the last expression evaluated is our return value 18 19 returnString = "JavaScript: (" + host_name + "):(" + port_no + 20 ") with (" + osname + "). DB Port = (" + port_db + ")." 21 </pre> 	<p>JavaScript sample:</p> <ul style="list-style-type: none"> • #1: Logs information to current logging file, incl. reading property 'LISA_PROJ_NAME' directly • #5: sets test state property <i>Hello</i> via TestExec object • #6: Reads String property 'LISA_HOST' via TestExec object with default string 'No Host' • #7: Sets var 'osname' via property 'os_name', which is in fact a system property 'os.name' • #8: Reads Integer property 'PORT' via TestExec object with default value of 9989 • #9: Adds to read port number to proof integer type • #10: Reads property DBPORT • #11: Adds to proof integer type. • #13: Sets property SCRIPT_OUT via TestExec object • #19: Last expression evaluated is returned by script
--	---

Scripts in Mobile Testing

Test cases for testing mobile application are part of DevTest Application Test. Test cases for mobile Applications can be customized by scripting using ‘Execute Script (JSR-223)’ test step the same way as for other applications.

Specifically for scripting Appium the ‘_webDriver’ variable is available since DevTest 8.0.1. This variable is a reference to the ‘MobileSession’ object associated with the test when running. Because of this, the ‘Test’ button at the bottom of the script editor is mostly useless as the ‘_webDriver’ variable is only active in a running session that is being recorded or played back. Otherwise errors are thrown when clicking this button.

‘_webDriver’ is an instance of class ‘org.openqa.selenium.remote’. Javadoc documentation of the API is available at org.openqa.selenium.remote.

Following sample executes on an iOS application:

```
// simulate a double click
var imgPath = "//UIAImage[@name='theImage']";
var imageElem = _webDriver.findElementByXPath(imgPath);
// double click simulation
imageElem.click();
imageElem.click();

// assert that it recognized the double tap
var actionPath = "//UIAStaticText[@name='Double Tap']";
var actionElem = _webDriver.findElementByXPath(actionPath);

// try to pinch the image on screen
var pinch = {
  startX: 335,
  startY: 338,
  endX: 333,
  endY: 474,
  duration: 1.8410
};
_webDriver.executeScript("mobile: pinchClose", pinch);
var actionText = actionElem.getText();
```

VSE Classes

Match Script and Scriptable Data Protocol Handlers work on transactions of virtual services. These scripts are supported by DevTest. DevTest supplies injected variables automatically giving access to the objects needed.

Request

‘lisa_vse_request’, ‘incomingRequest’ and ‘sourceRequest’ are supplied injected variables containing objects of class ‘com.itko.lisa.vse.stateful.model.Request’ and include single transaction requests. This class provides following methods:

String getOperation()	Returns the operation of the incoming request
void setOperation(String newOperation)	Changes the operation of the request
Boolean isBinary()	Returns whether or not the request body is binary or text data
void setBinary(Boolean bFlag)	Sets the ‘Binaryflag’ indicating whether or not the request body contains binary or text data
String getBodyText()	Returns the request body data as text
void setBodyText(String strBody)	Changes the request body to content of strBody.

	This method sets the BinaryFlag to 'false' automatically .
byte [] getBodyBytes()	Returns the request body data as binary
void setBodyBytes(byte [] arrayByte)	Changes the request body to content of arrayByte. This method sets the BinaryFlag to 'true' automatically .
String toString()	Returns the entire request as String
long id()	Returns the ID of the request

Response

'lisa_vse_response' is a supplied injected variable containing objects of class 'com.itko.lisa.vse.stateful.model.Response' and include a transaction response. This class provides following methods.

Boolean isBinary()	Returns whether or not the response's body is binary or text data
void setBinary(Boolean bFlag)	Sets the 'Binaryflag' indicating whether or not the response's body contains binary or text data
String getBodyText()	Returns the response body data as text
void setBodyText(String strBody)	Changes the response body to content of strBody. This method sets the BinaryFlag to 'false' automatically .
byte [] getBodyBytes()	Returns the response body data as binary
void setBodyBytes(byte [] arrayByte)	Changes the response body to content of arrayByte. This method sets the BinaryFlag to 'true' automatically .
String toString()	Returns the entire response as String
long id()	Returns the ID of the response
String getThinkTimeSpec()	Returns the time difference passed by between the request and this corresponding response, called 'Think time'
void setThinkTimeSpec()	Changes the 'Think time' of the response.

ParameterList

Transaction requests and responses contain arguments, attributes and Metadata. Access to each of them is provided by class 'com.itko.util.ParameterList'. A 'ParameterList' object is a set of key/value pairs. Methods of this class are available to both requests and responses:

ParameterList getArguments()	Retrieves the set of arguments from request or response
void setArguments (ParameterList args)	Changes the set of arguments
ParameterList getAttributes()	Retrieves the set of attributes from request or response
void setAttributes (ParameterList args)	Changes the set of attributes
ParameterList getMetaData()	Retrieves the Metadata from request or response
void setMetaData (ParameterList args)	Changes the Metadata
Boolean isDupesAllowed()	Indicates whether or not duplicate transactions are allowed in the Virtual Service
void setAllowDupes(boolean bFlag)	Sets the flag to indicate whether or not duplicate transactions are allowed in the Virtual Service.
void addParameter(String strKey, String strVal)	Sets a single parameter as key/value pair
void addParameters(String strParams)	Sets multiple parameters at once. Key and value are separated by '=', multiple key value pairs are concatenated by '&'.

	Sample: <code>p.addParameters("key1=val1&key2=val2");</code>
<code>String getParameterValue(String strKey)</code>	Returns the value of a parameter given by <i>strKey</i> .
<code>void setParameterValue(String strKey, String strVal)</code>	Changes value of parameter <i>strKey</i> to <i>strVal</i> .
<code>void removeParameter(String strKey)</code>	Removes parameter <i>strKey</i> from parameter list
<code>void clear()</code>	Removes all parameters from list

For additional methods available on ‘com.itko.util.ParameterList’ please see [5].

Security

You cannot call ‘System.Exit’ or ‘System.Exec’ from scripts. This is to prevent terminating the java process that runs the embedded script or running malicious scripts.

Logging

The injected ‘**_logger**’ variable logs data to the log file of the DevTest component that the script is executed within.

_logger

The injected variable ‘**_logger**’ is a SLF4J logger (<http://www.slf4j.org/manual.html>) with namespace ‘com.itko.lisa.script.logger’. Log level is set to INFO level by default, so a script can call something like `_logger.info("foo")` and that will show in the relevant log file (simulator.log, vse.log, workstation.log).

The log level for ‘com.itko.lisa.script.logger’ can be set to other values (e.g. DEBUG) in ‘logging.properties’. Then any calls to `_logger.debug("A debug message, my x value is {} and my y value is {}", x, y)` will be printed to the log file.

The default ‘logging.properties’ file does not contain a log level setting for ‘com.itko.lisa.script.logger’. So, the following line has to be added to `{{LISA_HOME}}/logging.properties` to enable different settings.

```
# for logging of scripts
log4j.logger.com.itko.lisa.script.logger=DEBUG
```

Possible log level values are:

- OFF – switches off logging
- ERROR – report error logs (`_logger.error()`) only. Errors have a serious impact to functionality
- WARN – report warning (`_logger.warn()`) and error logs only. Warnings can have an impact to expected functionality
- INFO – report informational (`_logger.info()`), warning and error logs only. Informational logs are for informational purposes only and are not supposed to have an impact to functionality
- DEBUG – report debug (`_logger.debug()`), informational, warning and error logs only. Debug logs are additional informational logs that should support error analysis.
- ALL – report any log, including trace logs (`_logger.trace()`). Trace logs are intended for extended diagnostic requests.

Sample 1

The following script snippet

```
_logger.error("ERROR: Sample script test - false");
_logger.warn("WARN: Sample script test - false");
_logger.info("INFO: Sample script test - false");
_logger.debug("DEBUG: Sample script test - false");
_logger.trace("TRACE: Sample script test - false");
```

And a setting of in `{{LISA_HOME}}/logging.properties`

```
# for logging of scripts
log4j.logger.com.itko.lisa.script.logger=ALL
```

Creates following log file output:

```
2015-01-21 13:49:03,770Z (14:49) [AWT-EventQueue-0] ERROR com.itko.lisa.script.logger - ERROR:Sample script test - false
2015-01-21 13:49:03,770Z (14:49) [AWT-EventQueue-0] WARN com.itko.lisa.script.logger - WARN: Sample script test - false
2015-01-21 13:49:03,771Z (14:49) [AWT-EventQueue-0] INFO com.itko.lisa.script.logger - INFO: Sample script test - false
2015-01-21 13:49:03,772Z (14:49) [AWT-EventQueue-0] DEBUG com.itko.lisa.script.logger - DEBUG:Sample script test - false
2015-01-21 13:49:03,773Z (14:49) [AWT-EventQueue-0] TRACE com.itko.lisa.script.logger - TRACE:Sample script test - false
```

Changing the setting in `{{LISA_HOME}}/logging.properties` to

```
# for logging of scripts
log4j.logger.com.itko.lisa.script.logger=INFO
```

Generates:

```
2015-01-21 14:08:38,253Z (15:08) [AWT-EventQueue-0] ERROR com.itko.lisa.script.logger - ERROR:Sample script test - false
2015-01-21 14:08:38,254Z (15:08) [AWT-EventQueue-0] WARN com.itko.lisa.script.logger - WARN: Sample script test - false
2015-01-21 14:08:38,255Z (15:08) [AWT-EventQueue-0] INFO com.itko.lisa.script.logger - INFO: Sample script test - false
```

Sample 2

The following script snippet demonstrates how to pass parameters of different types to a `'_logger'` call using `'{'` as place holders in the output string:

```
String p1 = "one";
int p2 = 2;
long p3 = 3;
double p4 = 4;
boolean p5 = true;
String[] p6 = {"five", "six"};
_logger.error("\nThis is an error message with parameters > {} {} {} {} {} {} <",
    p1, p2, p3, p4, p5, p6);
```

This leads to following output:

```
2015-02-09 12:09:09,518Z (13:09) [AWT-EventQueue-0] ERROR com.itko.lisa.script.logger -
This is an error message with parameters > one 2 3 4.0 true [five, six] <
```

Sample 3

Even complex objects can be logged. Following snippet logs content of injected variable 'testExec':

```
_logger.error("\nThis is testExec > {} <", testExec);
```

And generates following output:

```
2015-02-09 12:13:21.837Z (13:13) [AWT-EventQueue-0] ERROR com.itko.lisa.script.logger -
This is testExec > lisa.Execute script (JSR-223).rsp=true<BR>
EJBSERVER=localhost<BR>
LISA_PROJ_URL=file:/D:/DevTest-801GA/examples<BR>
JMSSCONNECTIONFACTORY=ConnectionFactory<BR>
LISA_DOC_PATH=D:/DevTest-801GA/examples/Tests<BR>
LISA_LAST_STEP=<BR>
lisa.designtime.testcaseinfo=com.itko.lisa.editor.TestCaseInfo@28952c1f<BR>
ENDPOINT1=http://localhost:8080/itkoExamples/EJB3UserControlBean<BR>
JNDIPOINT=1099<BR>
order.step.2.queue=queue/C<BR>
DBUSER=sa<BR>
DBNAME=itko_examples<BR>
user=webapp<BR>
LISA_RELATIVE_PROJ_URL=file:/D:/DevTest-801GA/examples<BR>
LISA_TC_PATH=D:/DevTest-801GA/examples/Tests<BR>
DBPORT=3306<BR>
lisa.hidden.scriptEngine.BeanShell=NotSerializableStateWrapper >> bsh.BshScriptEngine@966579c<BR>
robot=0<BR>
DBCONNURL=jdbc:derby://localhost:1529/lisa-demo-server.db<BR>
LISA_HOST=VOGUL01-W7<BR>
LISA_PROJ_NAME=examples<BR>
LISA_USER=admin<BR>
DBPASSWORD=sa<BR>
WSPORT=8080<BR>
instance=0<BR>
LIVE_INVOCATION_SERVER=localhost<BR>
EJBPORT=1099<BR>
PORT=8080<BR>
JNDIFACTORY=org.jnp.interfaces.NamingContextFactory<BR>
LISA_TEST_RUN_ID=EBCCBB4BB05111E492FF82F820524153<BR>
WSSERVER=localhost<BR>
testCaseId=EBCCBB4BB05111E492FF82F820524153<BR>
LISA_TC_URL=file:/D:/DevTest-801GA/examples/Tests<BR>
LISA_RELATIVE_PROJ_NAME=examples<BR>
LISA_RELATIVE_PROJ_ROOT=D:/DevTest-801GA/examples<BR>
testCase=Test Case<BR>
password=example-pwd<BR>
JNDIPROTOCOL=jnp<BR>
LIVE_INVOCATION_PORT=8080<BR>
DBDRIVER=org.apache.derby.jdbc.ClientDriver<BR>
user_prefix=csv_usertest<BR>
LISA_PROJ_PATH=D:/DevTest-801GA/examples<BR>
LISA_RELATIVE_PROJ_PATH=D:/DevTest-801GA/examples<BR>
LISA_PROJ_ROOT=D:/DevTest-801GA/examples<BR>
LISA_DOC_URL=file:/D:/DevTest-801GA/examples/Tests<BR>
SERVER=localhost<BR>
<
```

Events

'testExec' supports methods for pushing a TestEvent with an EVENT_LOGMSG Event ID:

- 'log()' – 'Log Message' event
- 'warn()' – 'Step Warning' event
- 'raiseEvent()' – custom event

testExec.log()

This is a convenience method for pushing a TestEvent with an EVENT_LOGMSG Event ID to the event system. This method comes with two different signatures:

- 'testExec.log(String shortMsg)'
- 'testExec.log(String shortMsg, String longMsg)'.

Following Code snippet applies both signatures:

```
testExec.log("Sample single script test - false");
testExec.log("Sample Log", "Sample script test - false");
```

Creates following events (when started in ITR)

Response	Properties	Test Events
Timestamp	EventID	Short Long
2015-01-21 15:19:00,054	Log message	Sample single script test - false N/A
2015-01-21 15:19:00,055	Log message	Sample Log Sample script test - false

testExec.warn()

This method raises a warning message into the events for this test. For scripts it supports two signatures:

- `testExec.warn(String shortMsg)`
- `testExec.warn(String shortMsg, String longMsg)`

The following Code snippet

```
testExec.warn("Sample short warning - false");
testExec.warn("Sample long warning", "Sample script test - false");
```

Creates following events (when started in ITR):

Timestamp	EventID	Short	Long
2015-01-21 15:28:51,417	Step warning	Sample short Warning - false	N/A
2015-01-21 15:28:51,417	Step warning	Sample long Warning	Sample script test - false

testExec.raiseEvent()

Previously described testExec methods raise specific events 'Log message' or 'Step warning' to the event system. Method `testExec.raiseEvent()` enables raising custom events with the event system.

Custom events must be specified in 'lisa.properties' before they can be used. Custom event numbers must start at 101. To define custom events 101 and 102, for instance, add following lines to 'lisa.properties':

```
#####
# DevTest Testing Stuff
#####
lisa.test.custevents=&101="Custom Event 101";&102="Custom Event 102"
```

There are multiple raiseEvent() signatures available. For details please see the Javadoc on the SDK [5].

The following one is recommended to use:

- `public void raiseEvent(int event,java.lang.String shortDesc, java.lang.String longDesc)`

Parameters have following meaning and effect:

1. event – this is the Event ID to use
2. shortDesc – the small-ish data associated with the event
3. longDesc – the long description can have much larger data associated

The following code snippet creates

```
testExec.raiseEvent(101,"Short Msg", "Long Message");
```

This output in the list of test events:

Timestamp	EventID	Short	Long
2015-01-23 14:36:21,443	Step history	C9E87A0DA30411E4B511764220524153	
2015-01-23 14:36:21,443	Step started	Execute script (JSR-223)~2	
2015-01-23 14:36:21,443	Step request	Execute script (JSR-223)~2	testExec.raiseEvent
2015-01-23 14:36:21,444	CUSTOM-101	Short Msg	Long Message
2015-01-23 14:36:21,444	Step response	Execute script (JSR-223)~2	true
2015-01-23 14:36:21,444	Step response time	Execute script (JSR-223)~2	1
2015-01-23 14:36:21,444	Assert evaluated	Execute script (JSR-223)~2 [Any Excep... The assertion of typ	
2015-01-23 14:36:21,444	Log message	Will execute the default next step	

Starting with DevTest 8.0.2 colors can be assigned to the custom events. This is optional but the color will be used by the workstation test event panel if is defined. The colors are based on hexadecimal encoding. For some examples please see http://en.wikipedia.org/wiki/Web_colors.

This feature can also be used to override the default colors in the event table.

With the above settings for custom events 101 and 102 and the additional settings in 'lisa.properties'

```
#=====
# DevTest Testing Stuff
#=====
# here we are assigning 'tomato' and 'mediumPurple' to events 101 and 102
lisa.test.custevents.colors=101=FF6347&102=9370DB
```

The following code snippet creates

```
testExec.raiseEvent(101,"short Msg", "long Message");
testExec.raiseEvent(102,"short Msg", "long Message");
```

This output in the list of test events:

Timestamp	EventID	Short	Long
2015-04-10 12:20:29,865	Step started	Execute script (JSR-223)	
2015-04-10 12:20:29,866	Step request	Execute script (JSR-223)	testExec.raiseEvent(101,"sh...
2015-04-10 12:20:29,866	Property set	lisa.hidden.scriptEngine.Bea...	NotSerializableStateWrapper...
2015-04-10 12:20:29,868	Custom Event 101	short Msg	long Message
2015-04-10 12:20:29,868	Custom Event 102	short Msg	long Message
2015-04-10 12:20:29,868	Step response	Execute script (JSR-223)	true
2015-04-10 12:20:29,868	Step response time	Execute script (JSR-223)	2
2015-04-10 12:20:29,868	Assert evaluated	Execute script (JSR-223) [An... The assertion of type "Asser...	
2015-04-10 12:20:29,868	Log message	Will execute the default next...	

Sharing Data

DevTest supports sharing data between its processes. Non-persistent data can be shared between VSMs on the same VSE. Persistent data are stored in the reporting database and can be shared across processes running with the same registry.

SharedModelMap

Sometimes non-persistent data needs to be shared across virtual services. The ‘SharedModelMap’ object supports this requirement. ‘SharedModelMap’ enables exchange of data by keys.

One VSM can do something like this (taken from scriptable DPH sample below):

```
com.itko.lisa.vse.SharedModelMap.put("transactionName", "currentOperation", fileToSave);
```

And another VSM can retrieve the key’s value as follows:

```
String fileName = com.itko.lisa.vse.SharedModelMap.get("transactionName", "currentOperation");
```

The first parameter in both calls is the optional namespace. It is optional but recommended so that keys are distinguished by different namespaces and unique within a namespace.

SharedModelMap supports following methods. For each method there is a variant available with and without specification of a namespace:

int size() int size(String nameSpace)	Returns the number of entries in the default or specified namespace, respectively.
boolean isEmpty() boolean isEmpty(String nameSpace)	Returns whether or not the (default or specified) namespace is empty.
boolean containsKey(String key) boolean containsKey(String nameSpace, String key)	Returns whether or not the (default or specified) namespace contains an entry with the given key.
boolean containsValue(Object key) boolean containsValue(String nameSpace, Object key)	Returns whether or not the (default or specified) namespace contains an entry with the given value.
Object getObject(String key) Object getObject(String nameSpace, String key)	Gets a value from the (default or specified) namespace.
String get(String key) String get(String nameSpace, String key)	Gets a value from the (default or specified) namespace cast as a string.
Object putObject(String key, Object value) Object putObject(String nameSpace, String key, Object value)	Puts a value into the (default or specified) namespace.
String put(String key, String value) String put(String nameSpace, String key, String value)	Puts a value into the (default or specified) namespace cast as a string, and returns the currently known value of the key cast as a String.
String remove(String key) String remove(String nameSpace, String key)	Removes a value from the (default or specified) namespace, and returns the currently known value of the key cast as a String.
void clear() void clear(String nameSpace)	Clears the (default or specified) namespace.
Set<String> keySet() Set<String> keySet(String nameSpace)	Returns the set of keys currently known in the (default or specified) namespace. The returned set will never be <i>null</i> .
void setCapacity(String nameSpace, int newCapacity)	Resizes the capacity of the specified namespace.

Each unique namespace is backed by a map that is restricted to a default capacity of 256 key/value pairs. Each namespace has a LRU (Last Recently Used) map created on demand the first time the namespace is used. It's backed by an ‘org.apache.commons.collections.map.LRUMap’. The LRU map determines which map entry to remove if there is no capacity left.

SharedModelMaps are not persisted across VirtualEnvironmentService restarts.

PersistentModelMap

DevTest (>7.1.1.141) provides a convenient mechanism to store arbitrary key/value pairs in its reporting database. The Registry provides a java API and it is also exposed as a RESTful service by the DevTest console.

Persistent maps are accessed by namespace and key. Values must be Strings. To store complex objects then the user is responsible for serialization using libraries such as XStream. By default key/value pairs in persistent model maps expire over time.

There are 4 basic APIS which are declared in PersistentMap and are implemented by 'com.itko.lisa.coordinator.TestRegistry':

String getMapValue(String nameSpace, String key)	Gets a value from the specified namespace cast as a String.
String putMapValue(String nameSpace, String key, String value)	Puts a value into the specified namespace, and returns the value of the key cast as a String.
String removeMapValue(String nameSpace, String key)	Removes a value from the specified namespace, and returns the value of the key cast as a String
Map<String,String> getAllMapValues(String nameSpace)	Returns all key/value pairs in a given namespace. Values are cast as Strings.

Restful API

Access to PersistentModelMap is also available by a RESTful API:

GET http://localhost:1505/lisa-invoke/persistentMap/namespace/key	Returns the value of 'key' in 'namespace'
GET http://localhost:1505/lisa-invoke/persistentMap/namespace	Returns all key/value pairs in namespace (equivalent to getAll)
PUT http://localhost:1505/lisa-invoke/persistentMap/namespace/key/value	Set 'key' in 'namespace' to 'value'
POST http://localhost:1505/lisa-invoke/persistentMap/namespace/key	Set 'key' in 'namespace' to requestBody
DELETE http://localhost:1505/lisa-invoke/persistentMap/namespace/key	Removes 'key' in 'namespace'.

Shared Data expiration

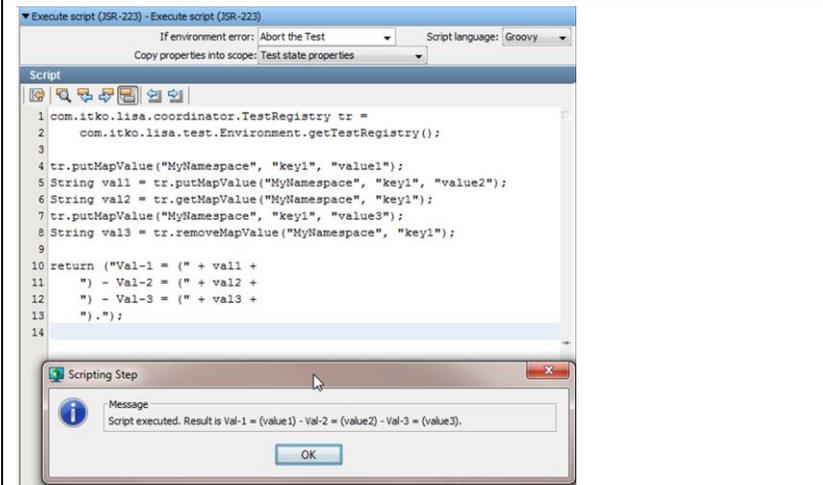
'key/value' pairs are deleted by the registry when they expire. The relevant data expiration properties are set in 'lisa.properties' files. The default values are as follows:

lisa.persistent.map.delete=true	whether to expire the data at all
lisa.persistent.map.delete.cycle=10m	check for expired entries every ten minutes
lisa.persistent.map.delete.age=30d	delete entries older than 30 days

The timestamps on key/value pairs are updated whenever an entry is added, updated or retrieved.

Sample

The following sample sets, retrieves and removes values to keys persistently to the reporting database:

 <pre> 1 com.itko.lisa.coordinator.TestRegistry tr = 2 com.itko.lisa.test.Environment.getTestRegistry(); 3 4 tr.putMapValue("MyNamespace", "key1", "value1"); 5 String val1 = tr.putMapValue("MyNamespace", "key1", "value2"); 6 String val2 = tr.getMapValue("MyNamespace", "key1"); 7 tr.putMapValue("MyNamespace", "key1", "value3"); 8 String val3 = tr.removeMapValue("MyNamespace", "key1"); 9 10 return ("Val-1 = (" + val1 + 11 ") - Val-2 = (" + val2 + 12 ") - Val-3 = (" + val3 + 13 ")."); 14 </pre> <p>Scripting Step</p> <p>Message</p> <p>Script executed. Result is Val-1 = (value1) - Val-2 = (value2) - Val-3 = (value3).</p> <p>OK</p>	<ul style="list-style-type: none"> • #1: create the persistentModelMap object • #4: set 'key1' to 'value1' • #5: set 'key1' to 'value2' and return previous value • #6: retrieve current value of 'key1' • #7: set 'key1' to 'value3' and return previous value • #8: remove 'key1' and return previous value
--	---

Debugging

Scripts cannot be debugged by an attached debugger. Using '`_logger`' for debugging scripts and breakpoints on custom java code called by the script might help.

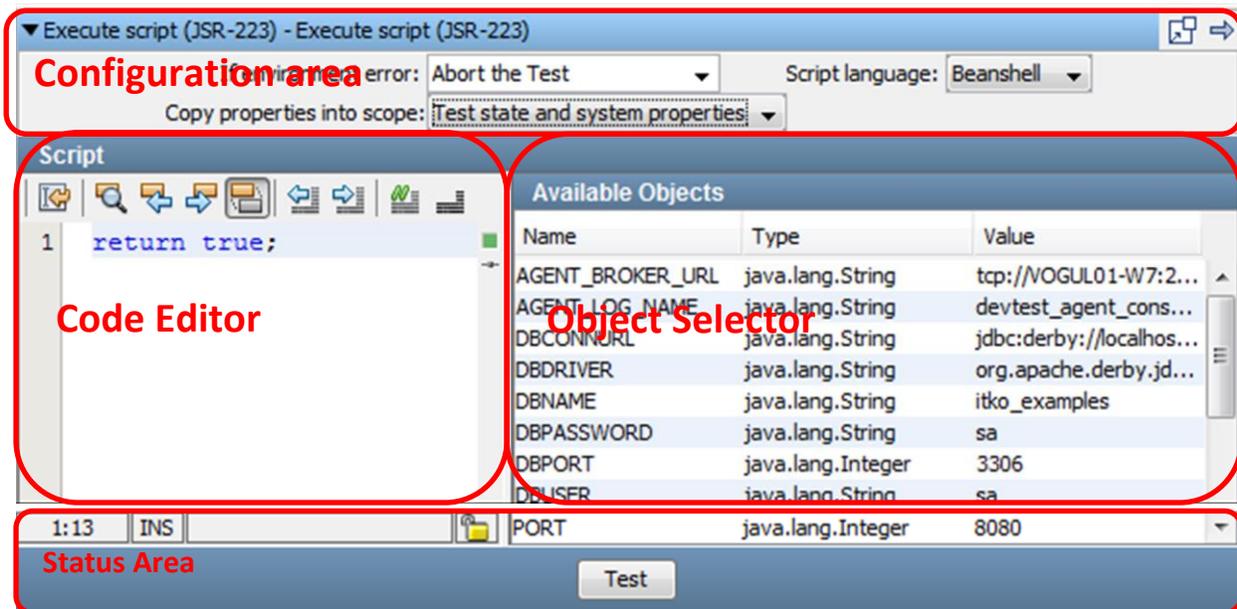
Best practice is to set the log level for scripting to DEBUG or ALL ([Logging](#)) and to use '`_logger.debug()`' when developing. When in production the log level is then set to INFO. With log level of INFO the data of script calls '`_logger.debug()`' will actually not be logged. Otherwise it would have a negative impact to performance.

The actual DevTest infrastructure that runs the script has a logger that you can set to DEBUG level – '`com.itko.lisa.test.ScriptExecHandler`'.

Editor

For all the scripted extensions there is a Script Editor in context available. The editor comprises of four sections

- Configuration area: specifies the language that will be used for scripting, the scope of properties that will be available for the script and context related effects of the script.
- Code editor – contains the script
- Object selector – lists the objects and properties available to the script
- Status/Action area – located at the bottom of the editor, containing information about cursor position in the editor, writing mode, read-only status, and – depending on the context a button to test the script.



Configuration Area

The configuration area includes specifications of the script’s context, how to react on the script’s result or a failure when executing the script.

Common to configuration areas of all script extensions are the specifications of the

- Scripting language
- Scope of available objects and properties

<p>Script language:</p> <p>Beanshell</p> <p>Beanshell</p> <p>Groovy</p> <p>JavaScript</p> <p>Velocity</p>	<p>Out of the box script languages</p> <ul style="list-style-type: none"> • Beanshell • Groovy • JavaScript • Velocity <p>Are available.</p> <p>Additional languages can be added on demand (Enabling additional Scripting Languages)</p>
<p>Copy properties into scope:</p>	<p>Specification of the scope of available properties and objects:</p>

		<p>'Copy properties into scope'</p> <ul style="list-style-type: none"> • Test state and system properties: all properties for the test case and system. This is the same scope as in previous product versions • Test state properties: properties that provide information about the test case • TestExec and logger only: only the TestExec and logger objects are available to the script (recommended)
--	--	--

Object Selector

Depending on the selection in drop-down list 'Copy properties into scope' the objects available for use in the script editor are listed in the 'Available Objects' panel to the right of the screen. The list includes primitive types of data (strings and numbers) and objects such as EJB response objects that were executed in the test case.

The drop-down list includes

- Test state and system properties: all properties for the test case and system. This is the same scope as in previous product versions
- Test state properties: properties that provide information about the test case
- testExec and logger only: only the testExec and logger objects are available to the script (recommended)

Notes:

- The less objects and properties are required by the script the better the performance to start the script.
- Properties can also be retrieved via testExec.

Double-click an entry in the Available Objects table to paste that variable name into the editor area.

<table border="1"> <thead> <tr> <th colspan="3">Available Objects</th> </tr> <tr> <th>Name</th> <th>Type</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>_logger</td> <td>org.slf4j.impl.Log4jLoggerAdapter</td> <td>org.slf4j.impl.Log4jLoggerAdap...</td> </tr> <tr> <td>testExec</td> <td>com.itko.lisa.test.TestExec</td> <td>lisa.Execute script (JSR-223).rs...</td> </tr> </tbody> </table>	Available Objects			Name	Type	Value	_logger	org.slf4j.impl.Log4jLoggerAdapter	org.slf4j.impl.Log4jLoggerAdap...	testExec	com.itko.lisa.test.TestExec	lisa.Execute script (JSR-223).rs...	<p>testExec and logger only</p>																								
Available Objects																																					
Name	Type	Value																																			
_logger	org.slf4j.impl.Log4jLoggerAdapter	org.slf4j.impl.Log4jLoggerAdap...																																			
testExec	com.itko.lisa.test.TestExec	lisa.Execute script (JSR-223).rs...																																			
<table border="1"> <thead> <tr> <th colspan="3">Available Objects</th> </tr> <tr> <th>Name</th> <th>Type</th> <th>Value</th> </tr> </thead> <tbody> <tr><td>DBCONNURL</td><td>java.lang.String</td><td>jdbc:derby://localhost:1529/li...</td></tr> <tr><td>DBDRIVER</td><td>java.lang.String</td><td>org.apache.derby.jdbc.Client...</td></tr> <tr><td>DBNAME</td><td>java.lang.String</td><td>itko_examples</td></tr> <tr><td>DBPASSWORD</td><td>java.lang.String</td><td>sa</td></tr> <tr><td>DBPORT</td><td>java.lang.Integer</td><td>3306</td></tr> <tr><td>DBUSER</td><td>java.lang.String</td><td>sa</td></tr> <tr><td>EJBPORT</td><td>java.lang.Integer</td><td>1099</td></tr> <tr><td>EJBSERVER</td><td>java.lang.String</td><td>localhost</td></tr> <tr><td>ENDPOINT1</td><td>java.lang.String</td><td>http://localhost:8080/itkoExa...</td></tr> <tr><td>JMSCONNECTIONFACTORY</td><td>java.lang.String</td><td>ConnectionFactory</td></tr> </tbody> </table>	Available Objects			Name	Type	Value	DBCONNURL	java.lang.String	jdbc:derby://localhost:1529/li...	DBDRIVER	java.lang.String	org.apache.derby.jdbc.Client...	DBNAME	java.lang.String	itko_examples	DBPASSWORD	java.lang.String	sa	DBPORT	java.lang.Integer	3306	DBUSER	java.lang.String	sa	EJBPORT	java.lang.Integer	1099	EJBSERVER	java.lang.String	localhost	ENDPOINT1	java.lang.String	http://localhost:8080/itkoExa...	JMSCONNECTIONFACTORY	java.lang.String	ConnectionFactory	<p>Available test state properties The screenshot shows a subset only</p>
Available Objects																																					
Name	Type	Value																																			
DBCONNURL	java.lang.String	jdbc:derby://localhost:1529/li...																																			
DBDRIVER	java.lang.String	org.apache.derby.jdbc.Client...																																			
DBNAME	java.lang.String	itko_examples																																			
DBPASSWORD	java.lang.String	sa																																			
DBPORT	java.lang.Integer	3306																																			
DBUSER	java.lang.String	sa																																			
EJBPORT	java.lang.Integer	1099																																			
EJBSERVER	java.lang.String	localhost																																			
ENDPOINT1	java.lang.String	http://localhost:8080/itkoExa...																																			
JMSCONNECTIONFACTORY	java.lang.String	ConnectionFactory																																			

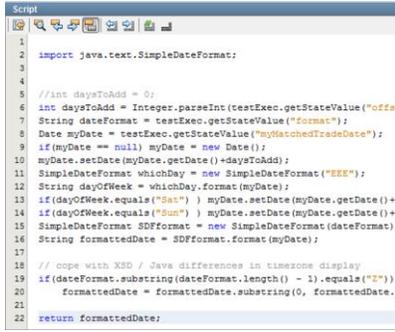
Available Objects		
Name	Type	Value
AGENT_BROKER_URL	java.lang.String	tcp://VOGUL01-W7:2009?dae...
AGENT_LOG_NAME	java.lang.String	devtest_agent_console_1301...
DBCONNURL	java.lang.String	jdbc:derby://localhost:1529/li...
DBDRIVER	java.lang.String	org.apache.derby.jdbc.Client...
DBNAME	java.lang.String	itko_examples
DBPASSWORD	java.lang.String	sa
DBPORT	java.lang.Integer	3306
DBUSER	java.lang.String	sa
EJBPORT	java.lang.Integer	1099
EJBSERVER	java.lang.String	localhost
ENDPOINT1	java.lang.String	http://localhost:8080/itkoExa...
java_awt_graphicsenv	java.lang.String	sun.awt.Win32GraphicsEnviro...
java_awt_printerjob	java.lang.String	sun.awt.windows.WPrinterJob
java_class_path	java.lang.String	D:\DevTest-800GA\install4j\4...
java_class_version	java.lang.Float	51.0
java_endorsed_dirs	java.lang.String	D:\DevTest-800GA\bin\..\lib\e...
java_ext_dirs	java.lang.String	d:\devtest-800ga\jre\lib\ext;C...
java_home	java.lang.String	d:\devtest-800ga\jre
java_io_tmpdir	java.lang.String	C:\Users\vogul01\AppData\Lo...
java_library_path	java.lang.String	D:\DevTest-800GA\bin\..\bin;...
java_net_preferIPv4Stack	java.lang.Boolean	true
java_protocol_handler_pkgs	java.lang.String	org.apache.axis.transport co...
java_rmi_server_randomIDs	java.lang.Boolean	true
java_runtime_name	java.lang.String	Java(TM) SE Runtime Environm...

Available test state and system properties
The screenshots show a small subset only

Code Editor

The code editor can be configured to display line numbers, a toolbar at the top and a status bar at the bottom.

	<p>Clicking on the left bar of the editor brings up a menu to add or remove</p> <ul style="list-style-type: none"> • Line numbers to the script • A menu bar on top of the editor • A status bar at the bottom of the editor
	<p>The Toolbar icons mean from left to right (Script Editor Toolbar)</p> <ul style="list-style-type: none"> • (greyed out) Returns you to the last edit that was made (Ctrl + Q) • Finds the next occurrence of the selected text (Ctrl + F3) • Finds previous occurrence (Shift F3) • Finds next occurrence (F3) • Toggles the highlight search (Alt+Shift+H) • Shifts the current line to the left four spaces (Alt+Shift+KeyPadLeft) • Shifts the current line to the right four spaces (Alt+Shift+KeyPadRight) • Inserts comments slashes (//) at the cursor position • Removes the comments slashes

 <pre> 1 2 import java.text.SimpleDateFormat; 3 4 5 //int daysToAdd = 0; 6 int daysToAdd = Integer.parseInt(testExec.getStateValue("offset")); 7 String dateFormat = testExec.getStateValue("format"); 8 Date myDate = testExec.getStateValue("myMatchedTradeDate"); 9 if(myDate == null) myDate = new Date(); 10 myDate.setDate(myDate.getDate()+daysToAdd); 11 SimpleDateFormat whichDay = new SimpleDateFormat("EEE"); 12 String dayOfWeek = whichDay.format(myDate); 13 if(dayOfWeek.equals("Sat")) myDate.setDate(myDate.getDate()+1); 14 if(dayOfWeek.equals("Sun")) myDate.setDate(myDate.getDate()+1); 15 SimpleDateFormat SDFformat = new SimpleDateFormat(dateFormat); 16 String formattedDate = SDFformat.format(myDate); 17 18 // cope with XSD / Java differences in timezone display 19 if(dateFormat.substring(dateFormat.length() - 1).equals("T")) 20 formattedDate = formattedDate.substring(0, formattedDate.length()-1); 21 22 return formattedDate; </pre>	<p>Editor for scripting.</p>
	<ul style="list-style-type: none"> • Status bar: On the bottom left hand side the cursor position is displayed, and whether the editor mode is set to INSert or OVerRide.

Status area

The status area contains the status bar information of the code editor and for some scripted extensions, such as the 'Execute Script (JSR-223)' test step a button to start a test of the script. For some scripted extensions, such as 'Scripted Assertion' this 'Test' button is integrated in the 'Configuration Area'.

Scripted Expression

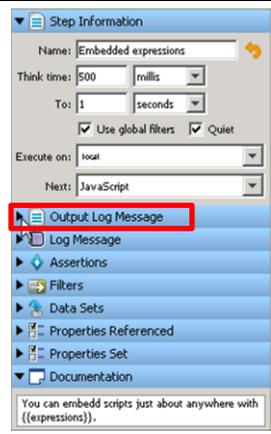
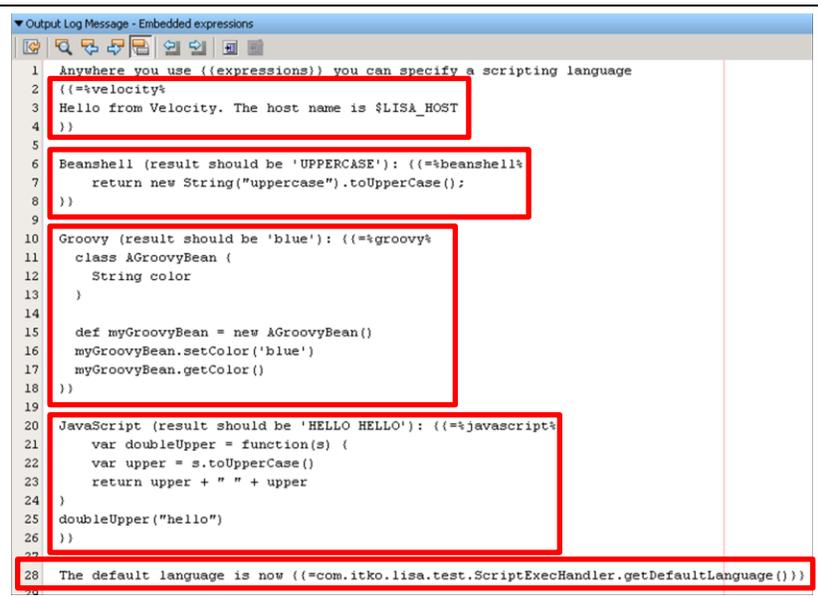
Expressions in DevTest, such as `{{=<expression>}}`, can be scripted. Using `{{=%<language specifier>%<script code>}}` the runtime environment starts the selected scripting engine to execute `<script code>`.

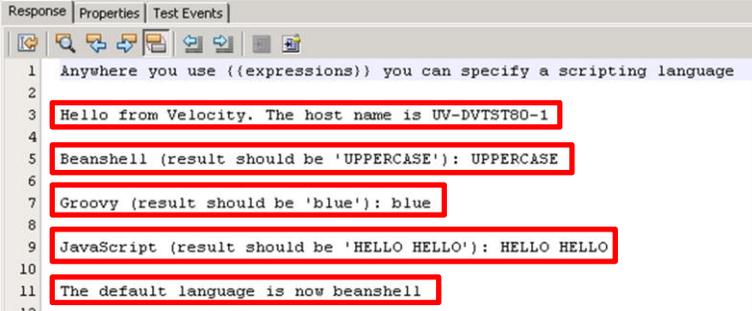
For the scripting languages available in DevTest out of the box the following language specifiers are defined

- BeanShell: `%beanshell%`, `%bsh%`
- Groovy: `%groovy%`, `%Groovy%`
- JavaScript: `%js%`, `%javascript%`, `%JavaScript%`, `%ECMAScript%`, `%ecmascript%`
- Velocity: `%velocity%`, `%Velocity%`

Sample

DevTest installation comes with a sample test case 'scripting.tst' in the 'examples' project. This test case includes a step 'Embedded expressions', which is an 'Output Log Message' step.

 <p>Step Information</p> <p>Name: Embedded expressions</p> <p>Think time: 500 millis</p> <p>To: 1 seconds</p> <p>Use global filters: <input checked="" type="checkbox"/> Quiet</p> <p>Execute on: local</p> <p>Next: JavaScript</p> <p>Output Log Message</p> <p>Log Message</p> <p>Assertions</p> <p>Filters</p> <p>Data Sets</p> <p>Properties Referenced</p> <p>Properties Set</p> <p>Documentation</p> <p>You can embed scripts just about anywhere with <code>{{(expressions)}}</code>.</p>	<p>To open the log message editor click on the 'Output Log Message' node in RHP</p>
 <p>Output Log Message - Embedded expressions</p> <pre> 1 Anywhere you use (expressions) you can specify a scripting language 2 {{=%velocity% 3 Hello from Velocity. The host name is \$LISA_HOST 4 }} 5 6 Beanshell (result should be 'UPPERCASE'): {{=%beanshell% 7 return new String("uppercase").toUpperCase(); 8 }} 9 10 Groovy (result should be 'blue'): {{=%groovy% 11 class AGroovyBean { 12 String color 13 } 14 15 def myGroovyBean = new AGroovyBean() 16 myGroovyBean.setColor('blue') 17 myGroovyBean.getColor() 18 }} 19 20 JavaScript (result should be 'HELLO HELLO'): {{=%javascript% 21 var doubleUpper = function(s) { 22 var upper = s.toUpperCase() 23 return upper + " " + upper 24 } 25 doubleUpper("hello") 26 }} 27 28 The default language is now {{=com.itko.lisa.test.ScriptExecHandler.getDefaultLanguage()}}</pre>	<p>This log snippet includes scripted expressions for each of the scripting engines available out of the box.</p> <p>The script section starts at <code>{{=%<language specifier>%</code> and ends at <code>}}</code>.</p> <p>Expression in line #28 returns the default scripting language as configured in <code>lisa.properties</code>.</p>

 <pre>1 Anywhere you use {{expressions}} you can specify a scripting language 2 3 4 Hello from Velocity. The host name is UV-DVTST80-1 5 Beanshell (result should be 'UPPERCASE'): UPPERCASE 6 7 Groovy (result should be 'blue'): blue 8 9 JavaScript (result should be 'HELLO HELLO'): HELLO HELLO 10 11 The default language is now beanshell 12</pre>	<p>After execution the scripted expression create the output at the left side.</p>
---	--

Test Step - Execute Script (JSR-223)

A test step is a workflow test case element that performs a basic action to validate a business function in the system under test. Steps can be used to invoke portions of the system under test. These steps are typically chained together to build workflows as test cases in the model editor. For each step, you can create filters to extract data or create assertions to validate response data. For details please see [Elements of a Test Step](#).

The 'Execute Script (JSR-223)' step lets you write and run a script to perform some function or procedure. This step has been introduced in DevTest 8.0 and replaces 'Java Script Step (deprecated)' ([Java Script Step \(deprecated\)](#)) going forward.

DevTest Product Documentation

- [1] - DevTest Solutions: Using the Workstation and Console with CA Application Test – [Elements of a Test Step](#)
- [2] - DevTest Solutions: Reference – [Execute Script \(JSR-223\) Step](#)
- [3] - DevTest Solutions: Reference – [Java Script Step \(deprecated\)](#)

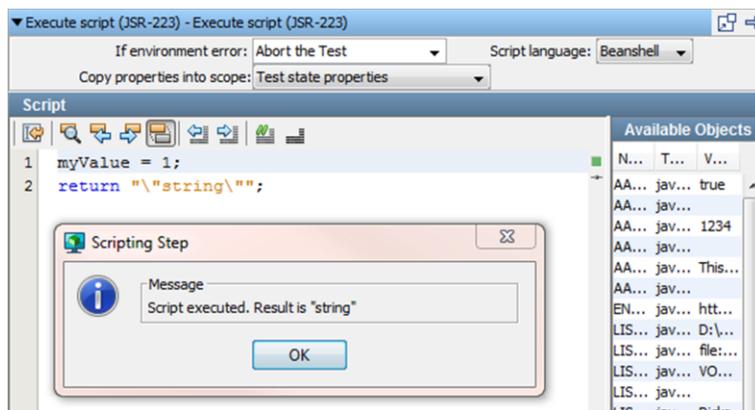
Input Parameters

Other than the selected injected variables and properties the script does not have any specific input variables. Input data are retrieved from properties, usually.

As mentioned before, to support test cases for mobile applications variable '_webDriver' is available since DevTest 8.0.1 to access the MobileSession object while the test is running. Please see [Scripts in Mobile Testing](#).

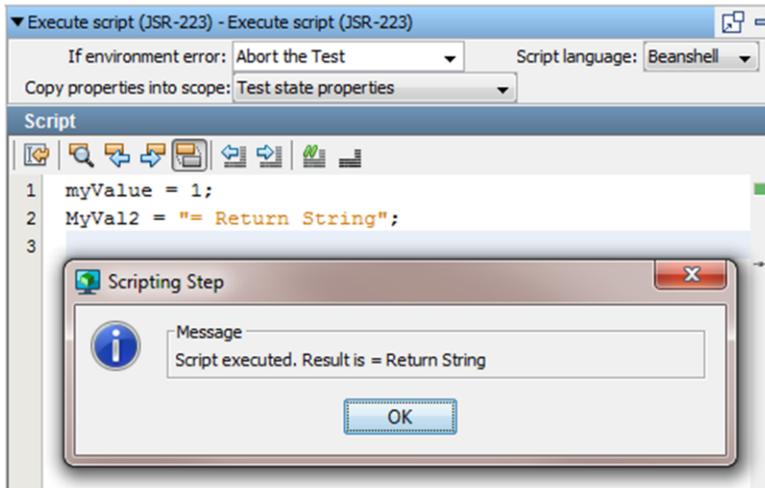
Output Parameters

It is best practice to supply a return value of scripts, if supported by the scripting engine.



As mentioned before, JavaScript does not support the concept of explicit return values in scripts. BeanShell and Groovy, however, do.

A script in 'Execute Script (JSR-223)' test steps does have to supply a specific return value. If no return value is specified the last evaluated expression is taken as the script's response. This applies to BeanShell, Groovy, and JavaScript.



By default, following properties are set when execution of the script is finished:

- 'LASTRESPONSE' – contains the script's return value
- 'lisa.<step name>.rsp' – also contains the script's return value

Other properties can be set by the script explicitly, using `'testExec.setStateValue()'`.

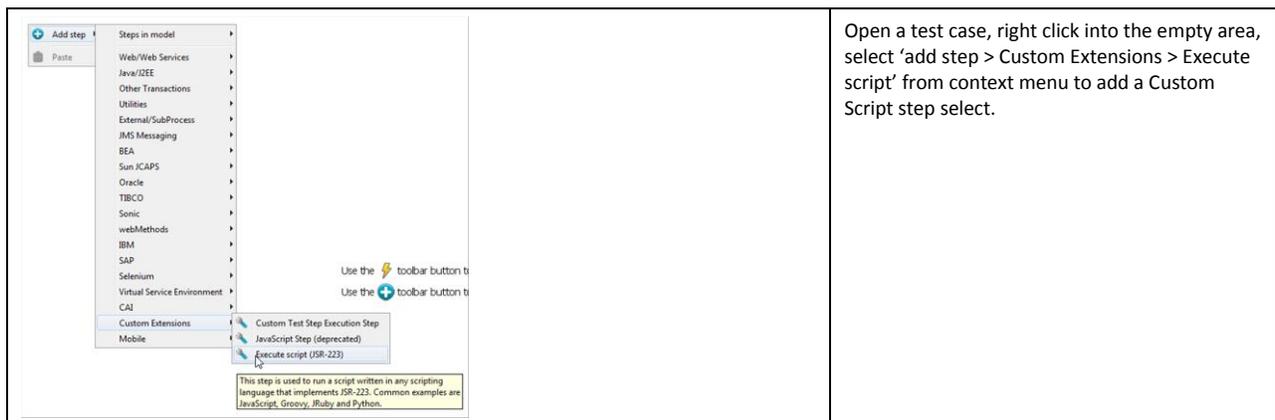
Logging output

The script can be started in different ways, which has an impact to the logging output

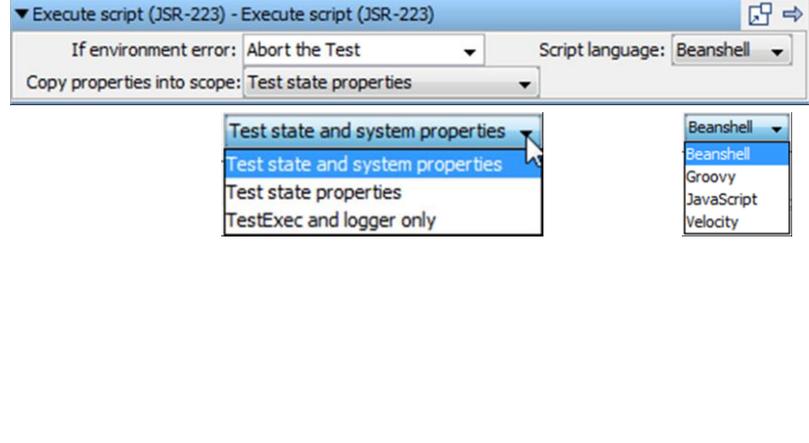
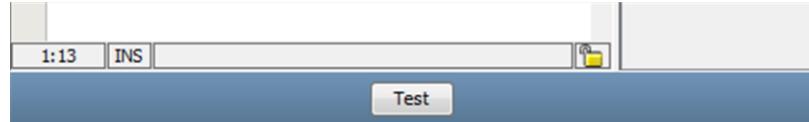
- Pressing the **'Test' button** in the Script Editor: '_logger' output is in **workstation.log**
- Starting the test case in **ITR**: '_logger' output is in **workstation.log**
- Submitting the test case to **'Stage a Quick Test'**
 - **Stage Local (No Coordinator Server)**: '_logger' output is in **workstation.log**
 - **Coordinator@Default**: '_logger' output is in **simulator.log**
- Submitting the test case to **'Stage Test'**: '_logger' output is in **simulator.log**

Editor

The editor to create a script opens in context of the 'Execute Script (JSR-223)' test step.



Open a test case, right click into the empty area, select 'add step > Custom Extensions > Execute script' from context menu to add a Custom Script step select.

	<ul style="list-style-type: none"> • 'If environment error' setting specifies the action to execute in case of a test step failure. • Language – determines the scripting language of the scripted assertion • 'Copy properties into scope' <ul style="list-style-type: none"> • Test state and system properties: all properties for the test case and system • Test state properties: properties that provide information about the test case • TestExec and logger only: only the TestExec and logger properties (recommended) • 'Run Assertion' – to open a window with the result of the script execution or a description of the errors that occurred.
	<p>At the bottom of the editor is a 'Test' button to test the script.</p>

Sample

This code sample takes a date, a date format and calculates a new date with a difference of a given amount of days. The difference of days can be positive or negative, and the new date must not be on a weekend. If it is on a weekend Monday after it is returned as the new date.

Input properties of this 'Execute Script (JSR-223)' test step are:

- 'myStartDate' (optional) – defines the date to start calculation from. Default value: today
- 'myOffset' (optional) – defines the number of days to add/subtract from start date. Default value: -8
- 'myFormat (optional)' – defines the date format to return the new date. Default value: 'M/d/yyyy', sample output: '2/9/2015' = Feb 2nd, 2015.

Output properties of this test step are:

- 'LASTRESPONSE' – contains a test step response by default
- 'lisa.<step name>.rsp' – also contains a test step response by default
- 'myNewDate' – contains output of this script, as it was set explicitly by the script code.

```

15 import java.text.SimpleDateFormat;
16
17 // read input parameters from properties
18 // use default values if properties are missing
19 int daysToAdd = testExec.getStateInt("myOffset", -8);
20 String strDateFormat = testExec.getStateString("myFormat", "M/d/yyyy");
21 Date myDate = testExec.getStateString("myStartDate", null);
22 // if no start date is given create a new one with date of today
23 if(myDate == null) myDate = new Date();
24 // Create a SimpleDateFormat object to format Date objects
25 SimpleDateFormat sdfFormat = new SimpleDateFormat(strDateFormat);
26 _logger.debug("Start date = ("+sdfFormat.format(myDate)+").");
27 // add the amount of days
28 Date newDate = new Date();
29 newDate.setDate(myDate.getDate()+daysToAdd);
30 _logger.debug("Calculated date = ("+sdfFormat.format(newDate)+").");
31 // Set the date format to retrieve the number of day of week:
32 // 1=Monday, 7=Sunday
33 SimpleDateFormat whichDay = new SimpleDateFormat("u");
34 // Retrieve the day of the week to check if it is on weekend.
35 // If so, add respective days to return Monday.
36 Integer dayOfWeek = Integer.parseInt(whichDay.format(newDate));
37 _logger.debug("Next day of the week is ("+whichDay.format(newDate)+").");
38 if(dayOfWeek == 6) newDate.setDate(newDate.getDate() + 2);
39 if(dayOfWeek == 7) newDate.setDate(newDate.getDate() + 1);
40 _logger.debug("Non-WE day of the week is ("+whichDay.format(newDate)+").");
41 // format the date in given format
42 String formattedDate = sdfFormat.format(newDate);
43 // Set Property to new date
44 testExec.setStateValue("myNewDate", formattedDate);
45 _logger.debug("New date ("+formattedDate+").");
46 // Return start date and new date.
47 return formattedDate;
48 // This return value will be available in properties LASTRESPONSE
49 // and lisa.<step name>.rsp for subsequent processing

```

- #15: import of java class needed for calculation
- #19 - #21: retrieve input properties with default values
- #23: if no start date is given take today's date
- #24: create a SimpleDateFormat object to format Date objects correctly
- #29: set the new Date object to the old one plus the difference of days
- #33: Specify the date format that returns the day of the week as number
- #36: convert the return string to an integer
- #38,#39: check for Saturdays and Sundays, and add additional days to end up on Monday
- #42: format the new date according to the requested format
- #44: set property 'myNewDate' to the new date
- #47: return the new date as the test step's response.

Scripted Assertion

The Scripted Assertion lets you write and run scripts to verify the expected outcome of a test step. The script's result must be a Boolean. Otherwise, false is returned.

DevTest Product Documentation

- [1] - DevTest Solutions: Using the Workstation and Console with CA Application Test – [Scripted Assertion](#)

Input Parameters

Other than the selected injected variables and properties the script does not have any specific input variables. Input data are retrieved from properties, usually.

Output Parameters

The script must return a Boolean value. Otherwise, false is returned.

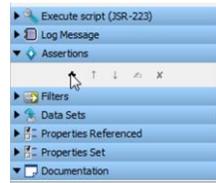
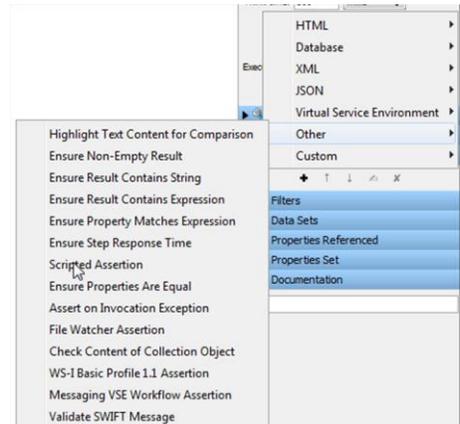
Logging output

The Scripted Assertion logs data to the same log file as the test case and its test steps do.

- Pressing the **'Test' button** in the Script Editor: **'_logger'** output is in **workstation.log**
- Starting the test case in **ITR**: **'_logger'** output is in **workstation.log**
- Submitting the test case to **'Stage a Quick Test'**
 - **Stage Local (No Coordinator Server)**: **'_logger'** output is in **workstation.log**
 - **Coordinator@Default**: **'_logger'** output is in **simulator.log**
- Submitting the test case to **'Stage Test'**: **'_logger'** output is in **simulator.log**

Editor

The script editor for a scripted assertion opens in context of a test step.

	<p>In the Test case, select a test step, expand the 'Assertions' on LHP and click the '+'-sign to open the context menus for the various assertions available out of the box.</p>
	<p>From context menu select 'Other > Scripted Assertion', which opens the editor</p>

	<ul style="list-style-type: none"> • Name – specifies the name of the assertion occurring in the list of assertions • ‘If’ clause specifies the condition of the script’s return value when the assertion will trigger • ‘Then’ defines the action to execute when the condition is met • Log – specifies the event text to print to the event log when the assertion triggers • Language – see Configuration Area ‘Copy properties into scope’ – see Configuration Area and Object Selector • ‘Run Assertion’ – to open a window with the result of the script execution or a description of the errors that occurred.
--	---

Sample

These samples show how to return Boolean expressions in Scripted Assertions for the different scripting languages. The Scripted Assertions samples are contained the ‘scripting.tst’ test case, which is part of the examples project of DevTest 8.0.

<pre>// This script should return a boolean result indicating // the assertion is true or false return "Greetings from beanshell".equals(LASTRESPONSE);</pre>	<p>This BeanShell sample is quite straight forward. It checks if property LASTRESPONSE equals the given static string. Return value is the result of this evaluation.</p>
<pre>// you can do anything with Groovy here. // as a quick demo let's use the 'tr' String function which is // exclusive to Groovy.. return LASTRESPONSE.tr("I", "i").equals("i'm flying!");</pre>	<p>This Groovy script similarly returns the result of the equality check of strings, leveraging the ‘tr’ method, which is special to Groovy. ‘tr’ translates characters from source set (“I”) to characters from replacement set (“i”), similar to the Unix ‘tr’ command’.</p>
<pre>// This script should return a boolean result // indicating the assertion is true or false _logger.info("JavaScript assertion") "6".equals(LASTRESPONSE)</pre>	<p>JavaScript does not have a concept of a ‘return’ statement outside of functions. The return value of a JavaScript based Scripted Assertion is the value of the expression evaluated last.</p>

Virtual Service Router Step

DevTest Service Virtualization supports multiple modes to execute a virtual service. Please see [VSE Console Request Events Details Tab](#) for details.

Within a Virtual Service Model this protocol-independent step routes a client request from a protocol-specific virtual service listen step to the protocol-specific response selector step or to the live invocation step, or to both. The decision is made based on the current execution mode for the running model.

Only when running in DYNAMIC mode, the actual mode is determined by a sub-process or a by a script. The return value of either one determines the actual mode virtual service model is running in. By default, a script is run. The default script evaluates to EFFECTIVE mode.

Currently, the step supports the BeanShell scripting engine only.

DevTest Product Documentation

- [1] - DevTest Solutions: Using Service Virtualization – [Virtual Service Router Step](#)

Input Parameters

Other than the selected injected variables and properties the script does not have any specific input variables. Input data are retrieved from properties, usually.

Output Parameters

This script must return either an 'enum' entry from class 'com.itko.lisa.vse.ExecutionMode' or a string that is the name of an enum entry. The DYNAMIC entry may not be returned. The script will be executed for DYNAMIC execution mode only.

Possible return values are:

Model Behavior	String	Enum
Most Efficient	"EFFICIENT"	ExecutionMode.EFFICIENT
Tracking	"TRACK"	ExecutionMode.TRACK
Live Invocation	"LIVE"	ExecutionMode.LIVE
Learning	"LEARNING"	ExecutionMode.LEARNING
Fail Over	"FAILOVER"	ExecutionMode.FAILOVER
Image Validation	"VALIDATION"	ExecutionMode.VALIDATION
Dynamic	"DYNAMIC"	ExecutionMode.DYNAMIC

Logging output

The 'Virtual Service Router' step script support the '*_logger*' variable. During replay logging output is part of VSE logging output and this is directed to vse.log.

Editor

This editor has not changed (yet) from previous versions. Currently it does not support a visual script language selector. However, the scripting language can be determined by a language specifier (e.g. %groovy%). The only object and property scope currently supported is ‘test state and system properties’.

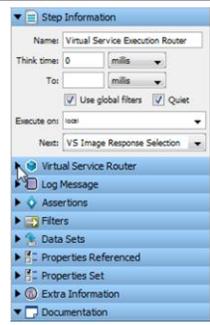
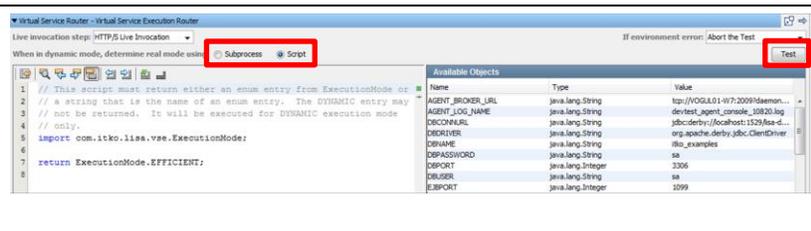
Setting scripting language

At the beginning of a script the scripting language can be configured by a language specifier %<scripting_language>%.

For the scripting languages available in DevTest out of the box the following language specifiers are defined

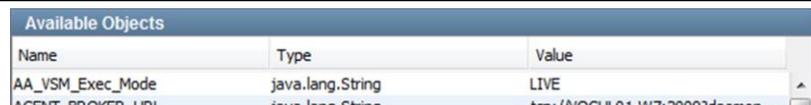
- BeanShell: %beanshell%, %bsh%
- Groovy: %groovy%, %Groovy%
- JavaScript: %js%, %javascript%, %JavaScript%, %ECMAScript%, %ecmascript%
- Velocity: %velocity%, %Velocity%

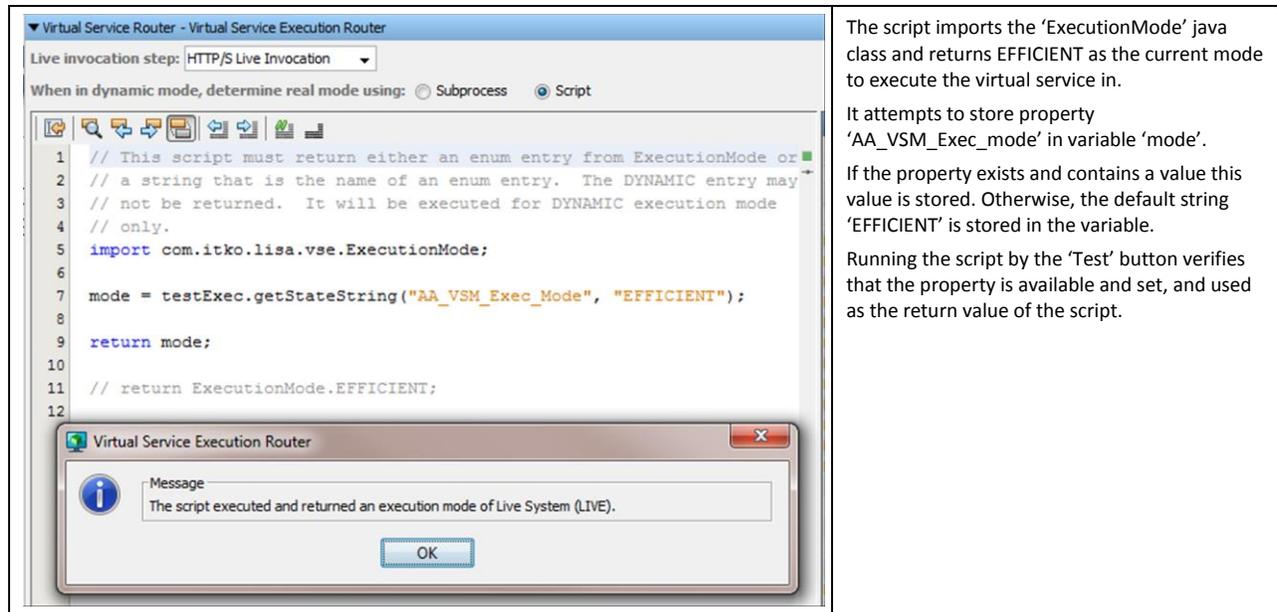
The language specifier must be the first statement in the script.

	<p>In the virtual service model, select the Virtual Service Router Step.</p> <p>In ‘Step information’ click ‘Virtual Service Router’ to open the associated script editor.</p> <p>This editor does not (yet) support language selection nor variable scoping.</p> <p>It supports BeanShell as scripting language only. All test states and system properties, plus injected testExec and _logger variables are available.</p>
	<p>There is a radio button to select whether the execution mode should be determined by a subprocess or by a script, which is the default selection.</p> <p>In the upper right corner a ‘Test’ button starts a script test.</p> <p>The remainder of the editor is the same as described in Editor.</p>

Sample

This sample script returns the value of a test state property, which contains the intended VSE execution mode for this virtual service model.

	<p>For demo purposes a project property ‘AA_VSM_Eexec_Mode’ is added with the value of ‘LIVE’.</p>
---	--



The screenshot shows the 'Virtual Service Router - Virtual Service Execution Router' window. The 'Live invocation step' is set to 'HTTP/S Live Invocation'. Under 'When in dynamic mode, determine real mode using:', the 'Script' radio button is selected. The main area contains a script with the following code:

```
1 // This script must return either an enum entry from ExecutionMode or
2 // a string that is the name of an enum entry. The DYNAMIC entry may
3 // not be returned. It will be executed for DYNAMIC execution mode
4 // only.
5 import com.itko.lisa.vse.ExecutionMode;
6
7 mode = testExec.getStateString("AA_VSM_Exec_Mode", "EFFICIENT");
8
9 return mode;
10
11 // return ExecutionMode.EFFICIENT;
12
```

Below the script, a 'Message' dialog box is displayed with the text: 'The script executed and returned an execution mode of Live System (LIVE)'. The dialog has an 'OK' button.

The script imports the 'ExecutionMode' java class and returns EFFICIENT as the current mode to execute the virtual service in.

It attempts to store property 'AA_VSM_Exec_mode' in variable 'mode'.

If the property exists and contains a value this value is stored. Otherwise, the default string 'EFFICIENT' is stored in the variable.

Running the script by the 'Test' button verifies that the property is available and set, and used as the return value of the script.

Match Script in Virtual Service Images

Match Scripts are used in VSE to identify a transaction in VSI that matches an incoming request. A Match Script is the mean to implement a custom matching algorithm that is not covered by the built-in matching algorithms, which come with DevTest 8.0 out of the box.

DevTest Product Documentation

- [1] - DevTest Solutions: Using Service Virtualization - [Match Script Editor](#)

Input Parameters

A Match Script has following injected variables

- *com.itko.lisa.vse.stateful.model.Request* **sourceRequest**: contains the recorded request object from VSI to compare against
- *com.itko.lisa.vse.stateful.model.Request* **incomingRequest**: contains the live request object received from service client.
- *com.itko.lisa.vse.RequestMatcher* **defaultMatcher()**: executes the default matching logic.

Please see [VSE Transaction Request](#) for available methods of objects 'incomingRequest' and 'sourceRequest'.

Access to arguments, attributes and metadata of transaction requests is handled by class 'ParameterList'. For available methods of this class please see [ParameterList](#).

Output Parameters

A Match Script must return a Boolean value to determine outcome of the matching calculation. A return value of 'true' means a matching transaction was found.

Instead of returning a Boolean value explicitly, the Match Script can call '*defaultMatcher.matches()*' to fall back to the default matching logic for further evaluation (see sample).

If there is an error evaluating the script, VSE deliberately ignores the error and defaults to the regular matching logic.

Logging Output

It is recommended to add logging and tracing into match script and to embed calls to the VSE matching logger. The VSE matching logger produces the messages in the *vse_XXX.log* file, where *xxx* is the service image name. For example, the log file of a virtual service 'kioskV6' appears as in the respective directory:

 <i>vse_matches.log</i>	27.01.2015 14:09	Text Document	1.581 KB
 <i>VS_kioskV6.log</i>	27.01.2015 13:40	Text Document	44 KB

The log level of INFO value typically reports every failure to match. If messages are logged at INFO level, later when the production settings are applied to the *logging.properties* file, the log level is set to WARN and log messages appear as a DevTest test event (a "Log Message" event), but do not appear in the log file.

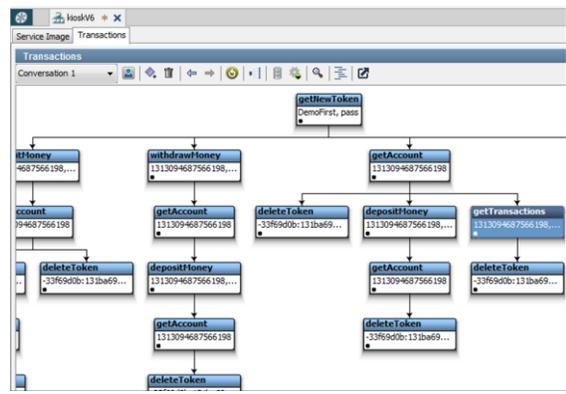
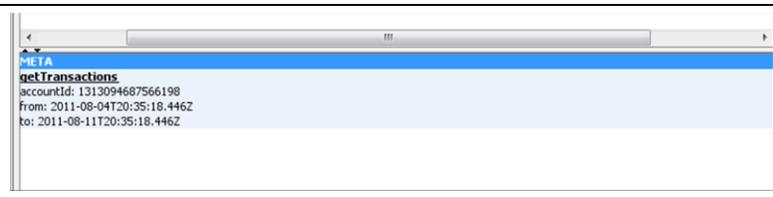
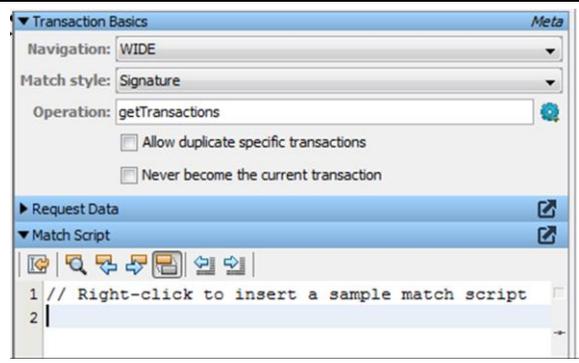
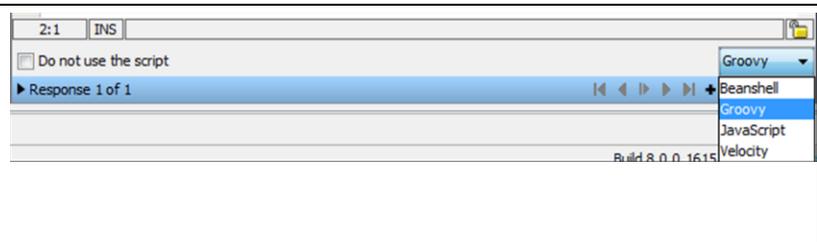
To simplify debugging, keep a separate log for VSE transaction match/no-match events. For production systems change INFO to WARN in following line of '*logging.properties*':

```
# Keep a separate log for VSE transaction match/no-match events, this makes debugging much easier.
# Change INFO below to WARN for production systems, the logging is expensive and will slow down
# systems with high transaction rates. Do not simply comment out the following line; explicitly
# set the log level to OFF or WARN instead of INFO
log4j.logger.VSE=INFO, VSEAPP
```

If you want to disable VSE logging, do not comment out this line, but set the property value to OFF.

Match Script Editor

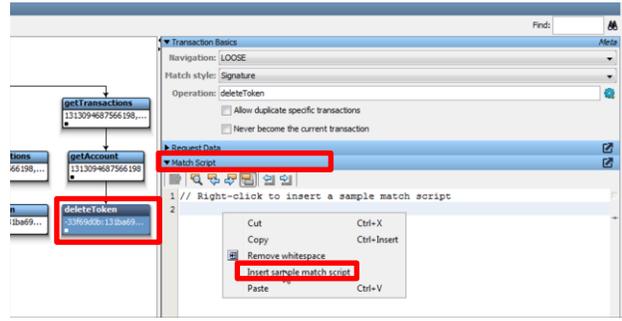
The Match Script editor is integrated into the virtual service image (VSI) editor. To open the Match Script Editor, open a virtual service image.

	<p>With the VSI Editor select the 'Transaction' tab. Pick and select a transaction,</p>
	<p>In the bottom part of the LHP select the response of the transaction that will host the Match Script. Each transaction response can have a different Match Script.</p>
	<p>In the RHP expand the 'Match Script' node of the selected transaction response Like any other of the script editors right click on the border of the editor and select any option to add a tool bar, line numbers and a status bar</p>
	<p>Apart from well-known information the status area of the editor includes the drop-down list to select the scripting engine. There is also a check box to deactivate the script. If deactivated the script is not called to identify the matching request, but the regular match algorithm is used instead.</p>

There is no button to run a local test of the script. Thus, scripts cannot be tested within context.

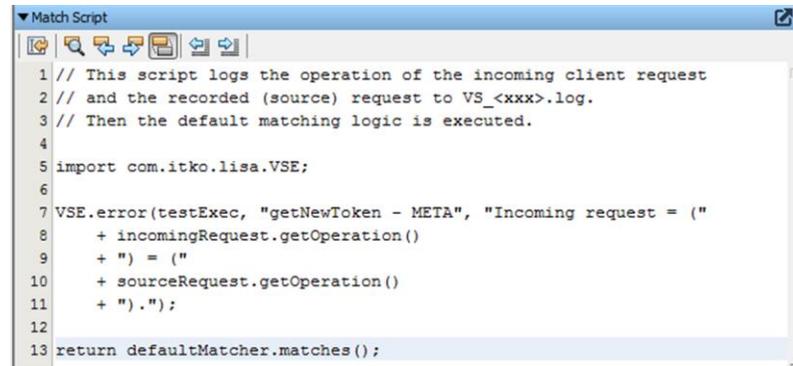
Sample 1

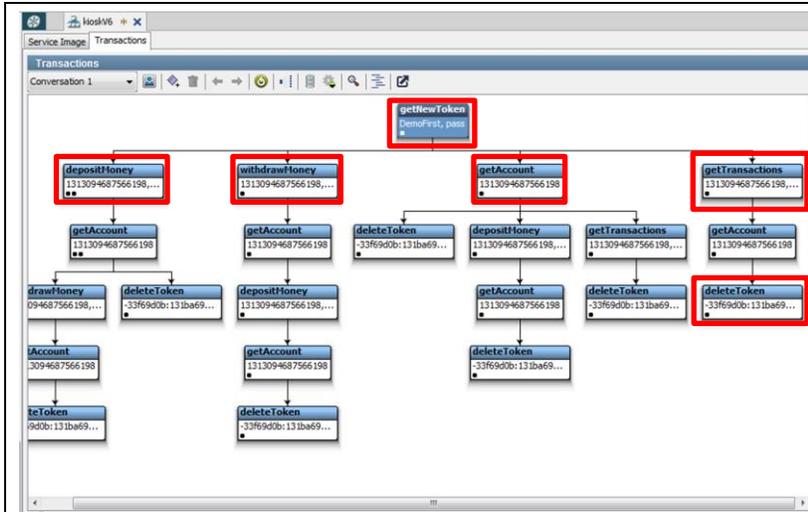
The editor includes an option to add a sample Match Script.

	<p>With the VSI Editor select the 'Transaction' tab. Pick and select a transaction, expand the 'Match Script' node.</p> <p>Right click into the editor area and select the 'Insert sample match script' menu</p>
<pre> 13 // This script only checks the operation name for a match, and to see if the 14 // first 3 characters of "SomeParameterName" match. 15 // Note that it pays no attention to match style / tolerance or comparison 16 // operators. 17 18 if (!incomingRequest.getOperation().equals("SampleRequest")) { 19 return defaultMatcher.matches(); 20 } 21 boolean operationsMatch = incomingRequest.getOperation().equals(sourceRequest.getOperation()); 22 if (operationsMatch) { 23 String incomingValue = incomingRequest.getArguments().get("SomeParameterName"); 24 String sourceValue = sourceRequest.getArguments().get("SomeParameterName"); 25 if (incomingValue.substring(0, 3).equalsIgnoreCase(sourceValue.substring(0, 3))) { 26 // true means these arguments match 27 return true; 28 } 29 } 30 // false means no match 31 return false; </pre>	<ul style="list-style-type: none"> • #18: This scripts first checks (#18) if the incoming live request from client includes an operation 'SampleRequest'. • #19: If it does not the default matching logic is executed by calling 'defaultMatcher.matches()'. • #21: If it includes an operation 'SampleRequest' then the operation name and the first three chars of parameter 'SomeParameterName' are compared from the live request and the recorded request. • #25: If both match true is returned to indicate the matched recorded request has been identified. • #31: if there is no match between live and recorded request false is returned.

Sample 2

The second sample uses the kioskV6 virtual service that is part of the examples project in DevTest workstation. The 'kioskV6.vsi' is extended by a Match Script that adds logging information to the recorded transactions to visualize the execution order.

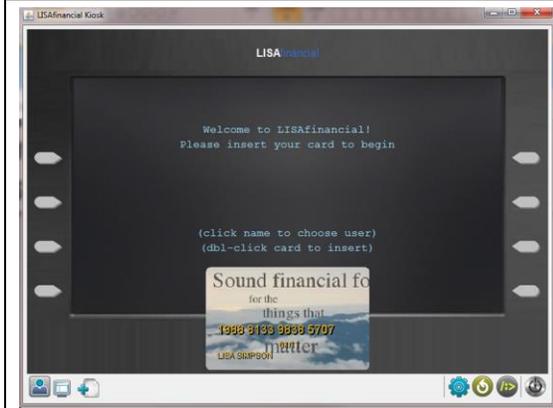
	<ul style="list-style-type: none"> • #7: the call logs the operation of the incoming client request (live) and the source (recorded) request. • This sample code identifies the META response of the 'getNewToken' transaction
---	--



META
getNewToken
 username: DemoFirst
 password: pass

Virtual Service Environment: VSE@Default

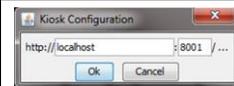
Name	Resource / Type	Status	Up-Time	Txn Count	Model Behavior	Group	Errors
kioskV6	8001 : http : //tkoExamples	Running	0:02:08	0	Most Efficient	scripting	



```
[kioskV6 [VS_kioskV6_Run]/1] ERROR - getNewToken - META Incoming request = (getNewToken) = (getNewToken).
[kioskV6 [VS_kioskV6_Run]/1] ERROR - getNewToken - Specific - 1 Incoming request = (getNewToken) = (getNewToken).
[kioskV6 [VS_kioskV6_Run]/1] ERROR - depositMoney - META Incoming request = (deleteToken) = (depositMoney).
[kioskV6 [VS_kioskV6_Run]/1] ERROR - withdrawMoney - META Incoming request = (deleteToken) = (withdrawMoney).
[kioskV6 [VS_kioskV6_Run]/1] ERROR - getAccount - META Incoming request = (deleteToken) = (getAccount).
[kioskV6 [VS_kioskV6_Run]/1] ERROR - getTransactions - META Incoming request = (deleteToken) = (getTransactions).
[kioskV6 [VS_kioskV6_Run]/1] ERROR - getNewToken - META Incoming request = (deleteToken) = (getNewToken).
```

The match script is added to each of the transactions META and specific definition.

Once the Match Scripts are completed the VS needs to be deployed and the transactions with Match Scripts in the service called.



Logging into the 'kiosk' sample application and logging out again produces the following log file content

The first two logs show the execution of the META and the specific match scripts for 'getNewToken', which is called upon log in. These log show that the operation match.

The remaining five logs are logs of the match scripts of the META definitions for all the transactions showing that the operations do not match.

Scriptable Data Protocol

A data protocol defines the payload of a transport protocol. The Data Protocol Handler (DPH) is responsible for parsing and translating requests from live service client or responses from live or virtual services, or both of them.

The Scriptable Data Protocol is available for situations where you need a small amount of processing on the live request, the live or virtual response, or both when recording or playing back a Virtual Service. DPHs can be 'daisy-chained', meaning the output of one DPH is passed as input to the next one.

Note: It needs to be verified if Scriptable DPHs support 'daisy-chaining'. Current assumption is that Scriptable DPHs do not, but are always executed first.

DevTest Product Documentation

- [1] - DevTest Solutions: Using Service Virtualization - [Scriptable Data Protocol](#)

Scripts

The Scriptable Data Protocol comprises of three scripts:

- Request script – to make data from client requests available for processing by DevTest
- Live Response script – to make data from live service responses available to DevTest processing
- Playback/Virtual Response script – to format the response from Virtual Service to suit the client's response requirements.

Request Script

This script is to parse the incoming request and generate the entries DevTest needs to identify the incoming client request. At minimum, it will need to define the name of the operation and add parameters for DevTest to match against. This script is used during recording and playback.

Response Script for generation of Virtual Service

DevTest can use text response payloads, binary ones, JSON ones or XML ones. The most human-readable of these is often XML, and it is most integrated with DevTest auto-correlation and time-shifting facilities (magic dates and magic strings) so the purpose of the generation response script is to parse the response from live service and store it as a set of XML values that can be viewed by testers and developers in DevTest Workstation.

Response Script for playback of Virtual Service

The response will need to be parsed from how DevTest Workstation has displayed it above. This script is to take the mashed response and reformat it back to the format that the client application requires.

Input Parameters

The request script has access to injected variable 'lisa_vse_request'. This variable gives access to elements of the client request. Please see [VSE Transaction Request](#) for available methods.

The response scripts have access to injected variable 'lisa_vse_response'. This variable gives access to elements of the live or the virtual response. Please see [VSE Transaction Response](#) for available methods.

Access to arguments, attributes and metadata of requests and responses is handled by class 'ParameterList'. For available methods of this class please see [ParameterList](#).

Output Parameters

There is no specific output parameter required.

Logging output

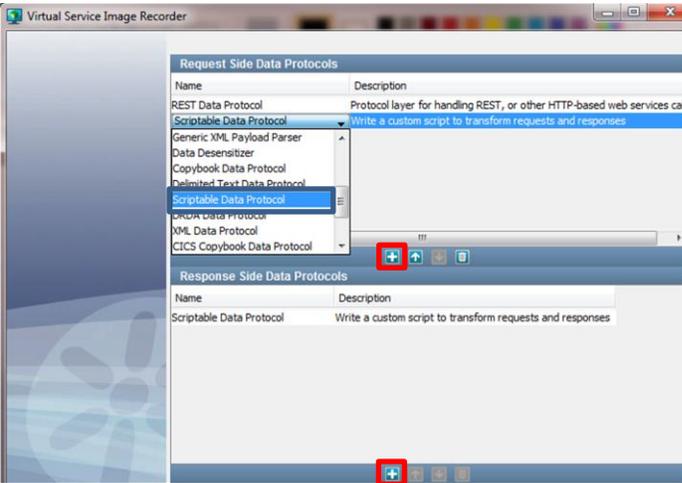
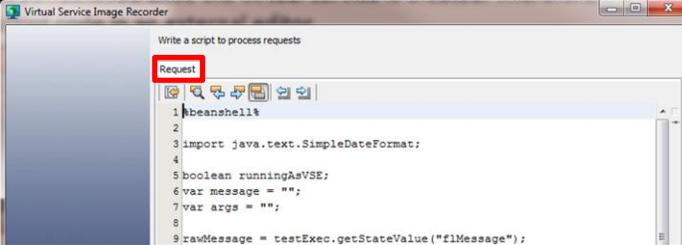
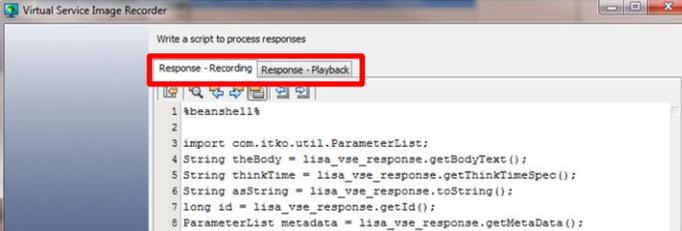
The DPH scripts support the ‘*_logger*’ variable. During recording logging output is part of the workstation log.

When deployed to VSE logging output is directed to vse.log.

Editor

Scripted Data Protocol Handlers are created when configuring the Service recorder in DevTest Workstation. Once the service recording is completed, data protocol handlers are configured for the request, the live and the playback responses.

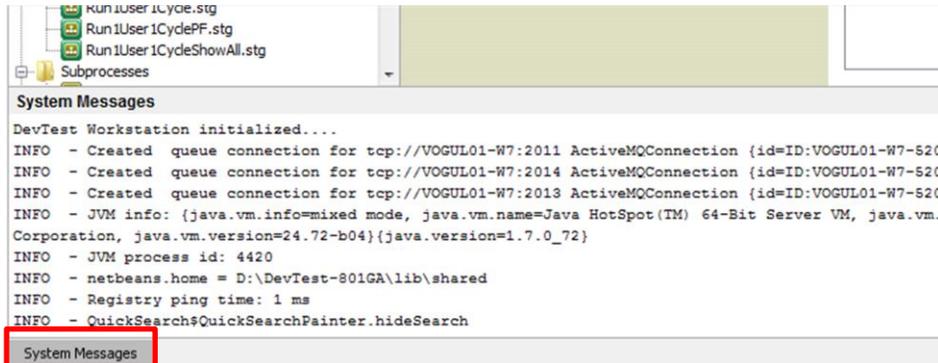
Note: Unless the recorder setup is saved at the very last step of the recorder wizard the code of scripted DPH cannot be retrieved once the virtual service is created. Therefore it is strongly recommended to save DPH script code in an external editor.

	<p>Clicking on the ‘+’ button adds a new row of Data Protocols to each of the lists.</p> <p>From dropdown list the ‘Scriptable Data Protocol’ is available</p>
	<p>The next screen opens the script editor for the request-side Scriptable DPH.</p>
	<p>The next screen opens the script editor for the both response-side Scriptable DPHs.</p>

Best Practices

Developing a DPH is cumbersome with regard to testing and debugging. First and foremost recommendation for debugging scripts is always to set the log level for scripts to DEBUG or ALL (see [Logging](#)). Second recommendation is then to take advantage it by making extensive use of

'`_logger.debug("{}"; value)`' statements. Check 'workstation.log' for exceptions and logging output and expand the DevTest System Message panel from the bottom of the Workstation window by double click on 'System Messages'

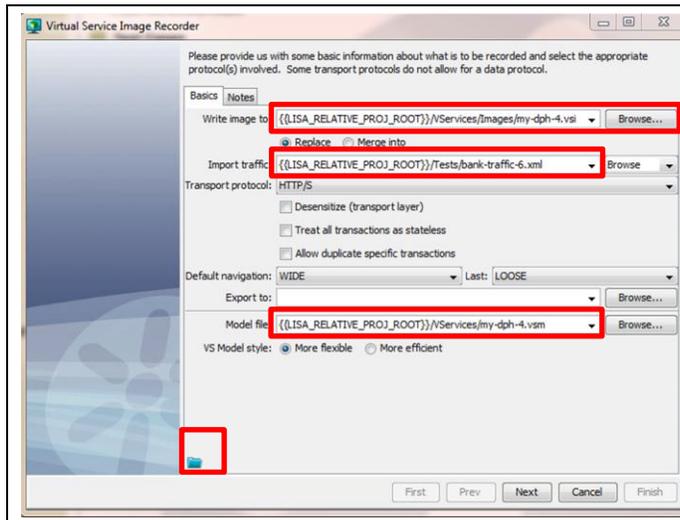


DPH Test using 'Execute Script (JSR-223)

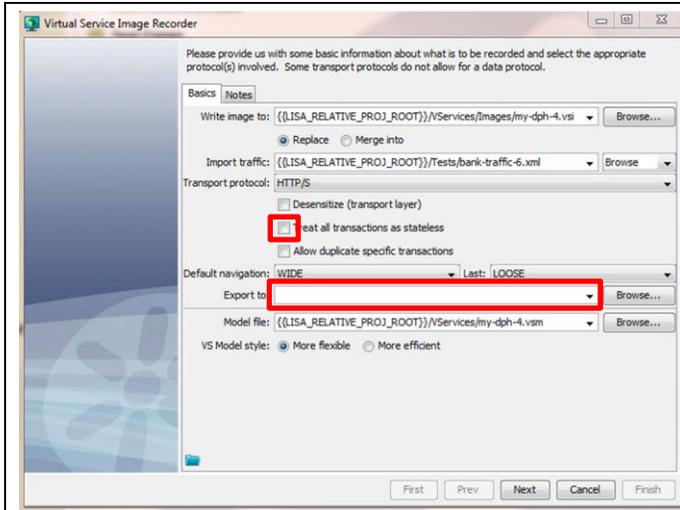
A recommended approach is to test the DPH script code with an 'Execute Script (JSR-223)' test step if possible. Sample 1 below demonstrates how code would look like that can be used in both DPH and in a test case. Main difference is that when used in a test case the transaction request or response is read from properties instead. VSE passes transaction request and responses in by injected variables.

DPH Test within VSE Recorder

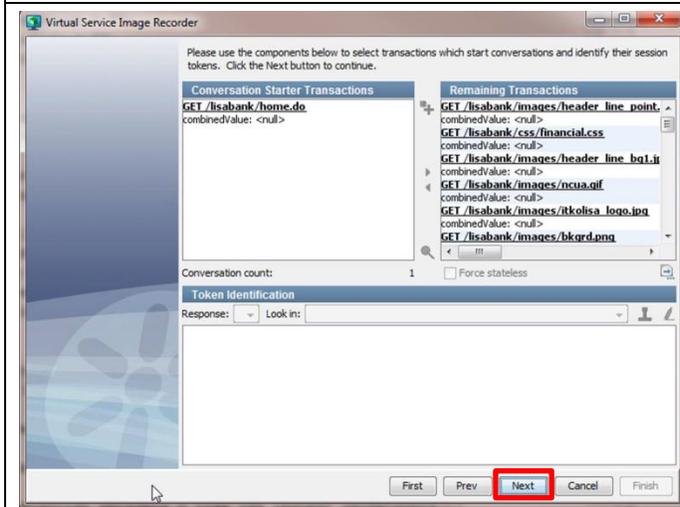
Following recommendations might help when testing the DPH within the VSE recorder:



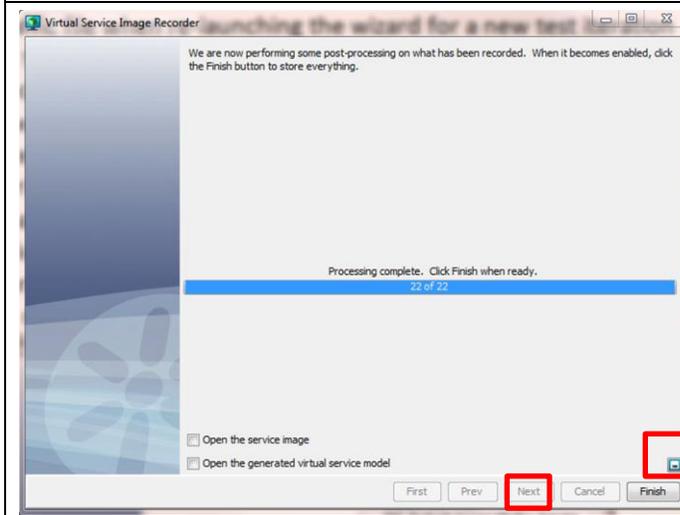
1. Load the saved VSE recorder settings for the next test cycle upon re-launch of the recorder wizard.
2. Following parameters are not set when loading a .vrs file
 - a. 'Write image to'
 - b. 'Import traffic'
 - c. 'Model file'
3. Test with traffic that was recorded beforehand.
 - a. Import the recorded traffic via 'Import traffic' combo box.



4. For debugging purposes keep parameter 'Treat all transactions as stateless' unchecked. This will make sure to have an intermediate wizard step available to check for exceptions reported by the scripting engine, and to return to the script editor.
5. Make sure not to override the existing raw traffic file when re-launching the wizard for a new test cycle. Otherwise the traffic file might contain data modified by the DPHs.
 - a. Make sure list box 'Export traffic' is empty or has a different file path than 'Import traffic'



6. Prior to clicking 'Next' on the screen below check for any exceptions in workstation.log, particularly for exceptions of scripting engines.
7. **After pressing the 'Next' button there is no way to return to the script editors**



8. At the end of the wizard save the recorder settings in a .vrs file to save the latest DPH script code.
9. Always have a backup copy of the latest Scriptable DPH version in an external editor.
10. In case the latest recorder settings are not saved in a .vrs file there is no way to retrieve a copy of the latest script source code

Errata

Out of the box in DevTest 8 there is sample code added to each of the DPHs. However, with current DevTest 8 versions and latest LISA versions method 'addParameter(String key, String value)' is **not supported**.

```
// Adding parameters
p.addParameters("key1=val1&key2=val2"); // many at once
p.addParameter("key3", "val3"); // one at a time
```

Javadocs ([5]) for class 'ParameterList' does not contain a method with this signature.

Please see Sample 2 below how to use 'addParameter(String label, String key, String value)' instead.

Sample 1

This DPH sample records the request sent from client and the live responses sent by the live services in a file. The request-side DPH exchanges data with the response-side DPH using a 'SharedModelMap'.

Recorded requests and responses are logged in individual files. File names comprise of the operation's name, the ID, a timestamp to make them unique, and an indicator whether the file contains a request or response.

Request DPH

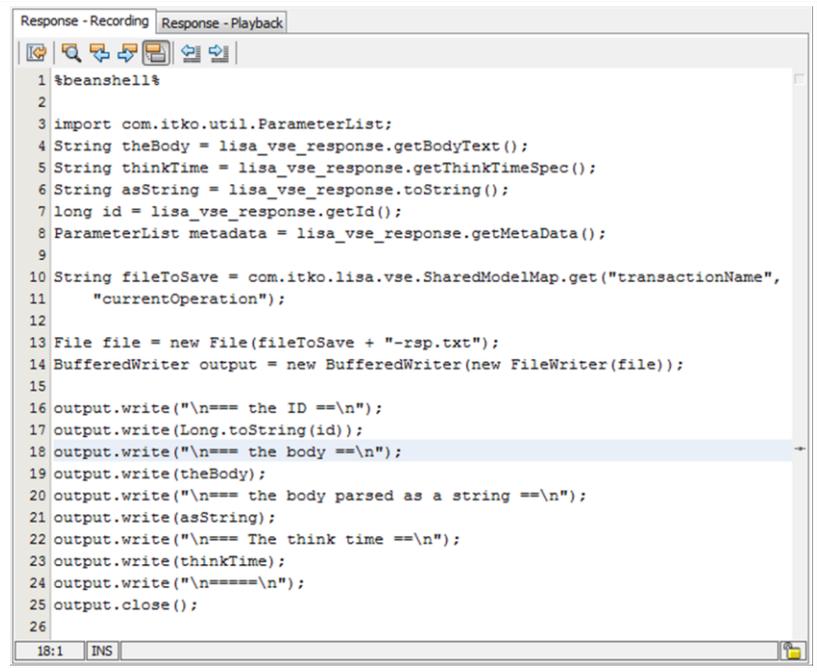
The following sample DPH logs requests from live client system. In order to test and debug main parts of the Scriptable DPH in a test step, the script can either retrieve the client request from VSE or from a property.

<pre>Request 1 %beanshell% 2 3 import java.text.SimpleDateFormat; 4 5 boolean runningAsVSE; 6 var message = ""; 7 var args = ""; 8 9 rawMessage = testExec.getStateValue("flMessage"); 10 if (rawMessage == null) { 11 rawMessage = lisa_vse_request.getBodyBytes(); 12 args = lisa_vse_request.getArguments(); 13 runningAsVSE = true; 14 } else { 15 runningAsVSE = false; 16 } 17</pre>	<ul style="list-style-type: none"> • #9: retrieval of message from property • #10-#16: if retrieval from property fails it means that the script is not run in a test step but as part of a VSM in VSE. If retrieval fails message content and arguments of operation are retrieved from current request in VSE • A property 'runningAsVSE' is set accordingly
--	---

<pre> 18 String operation = ""; 19 String theBody = ""; 20 String asString = ""; 21 long id; 22 23 if(runningAsVSE) { 24 import com.itko.util.ParameterList; 25 26 operation = lisa_vse_request.getOperation(); 27 isBinary = lisa_vse_request.isBinary(); 28 29 byte[] b; 30 if(isBinary) { 31 b = lisa_vse_request.getBodyBytes(); 32 } else { 33 theBody = lisa_vse_request.getBodyText(); 34 } 35 36 asString = lisa_vse_request.toString(); 37 id = lisa_vse_request.getId(); 38 39 ParameterList args = lisa_vse_request.getArguments(); 40 41 ParameterList attributes = lisa_vse_request.getAttributes(); 42 43 ParameterList metadata = lisa_vse_request.getMetaData(); 44 } else { </pre>	<ul style="list-style-type: none"> • #23ff: If running in VSE message properties are retrieved such as <ul style="list-style-type: none"> ○ Operation ○ Payload as binary or text depending on content ○ ID ○ Request in String representation ○ Operation's arguments, attributes and Meta data
<pre> 44 } else { 45 flMessage = testExec.getStateString("flMessage", ""); 46 theBody = flMessage; 47 asString = flMessage; 48 operation = "myOperation"; 49 id = 0001; 50 51 } </pre>	<ul style="list-style-type: none"> • #44ff: If NOT running in VSE message <ul style="list-style-type: none"> ○ Operation is hard coded as 'myOperation' ○ Payload is text ○ ID is hardcoded as '0001'
<pre> 52 operation = operation.replaceAll("/", "_"); 53 54 SimpleDateFormat timestampFormat = new SimpleDateFormat(55 "yyyyMMdd_HHmssSSS"); 56 Date currentTimestamp = new Date(); 57 String myDate = timestampFormat.format(currentTimestamp); 58 59 proj_root = testExec.getStateValue("LISA_PROJ_ROOT"); 60 fileToSave = proj_root + "/data/" + operation + "_" + id + "_" + myDate; 61 com.itko.lisa.vse.SharedModelMap.put("transactionName", 62 "currentOperation", fileToSave); 63 64 File file = new File(fileToSave + "-req.txt"); 65 BufferedWriter output = new BufferedWriter(new FileWriter(file)); 66 output.write("\n=== The Operation ==\n"); 67 output.write(operation); 68 output.write("\n=== the ID ==\n"); 69 output.write(Long.toString(id)); 70 output.write("\n=== the body ==\n"); 71 output.write(theBody); 72 output.write("\n=== the body parsed as a string ==\n"); 73 output.write(asString); 74 output.write("\n====="); 75 output.close(); 76 </pre>	<ul style="list-style-type: none"> • #52ff: Web service requests and responses can contain forward slashed ('/'), which may interfere with file path definitions. So they are replaced by underscores ('_') • #54ff: calculation of a timestamp • #59ff: calculation of the file path prefix, comprising of the DevTest project path, its '/data/' subdirectory, the operation's name, the message id, and the time stamp. • #61: The file path prefix is stored in Shared Model Map. • #64: The file name is constructed based on the file path prefix and the suffix that determines whether it is a request or a response. On the request side DPH the suffix '-req.txt' is added. • #65ff: retrieved message content is printed to the file.

Response DPH

The following sample DPH logs responses from live service.

 <pre> 1 %beanshell% 2 3 import com.itko.util.ParameterList; 4 String theBody = lisa_vse_response.getBodyText(); 5 String thinkTime = lisa_vse_response.getThinkTimeSpec(); 6 String asString = lisa_vse_response.toString(); 7 long id = lisa_vse_response.getId(); 8 ParameterList metadata = lisa_vse_response.getMetaData(); 9 10 String fileToSave = com.itko.lisa.vse.SharedModelMap.get("transactionName", 11 "currentOperation"); 12 13 File file = new File(fileToSave + "-rsp.txt"); 14 BufferedWriter output = new BufferedWriter(new FileWriter(file)); 15 16 output.write("\n=== the ID ==\n"); 17 output.write(Long.toString(id)); 18 output.write("\n=== the body ==\n"); 19 output.write(theBody); 20 output.write("\n=== the body parsed as a string ==\n"); 21 output.write(asString); 22 output.write("\n=== The think time ==\n"); 23 output.write(thinkTime); 24 output.write("\n=====\n"); 25 output.close(); 26 </pre>	<ul style="list-style-type: none"> • #4ff: retrieval of response content, including payload, meta data and – specific for the response – the think time, as required by the live service. • #10: the file path prefix is get from SharedModelMap where it was stored by the Request DPH. • #13: For the response log file the file path name is completed yb the ‘-rsp.txt’ suffix • #14ff: logging of the response data.
--	---

Sample 2

The following Scriptable DPH example extends the list of arguments of an operation by an additional argument:

```
%beanshell%

import com.itko.util.ParameterList;
import com.itko.util.Parameter;

// Retrieve transaction operation and arguments
_logger.debug("{} ", lisa_vse_request.getOperation());
_logger.debug("All Arguments:");
ParameterList args = lisa_vse_request.getArguments();
_logger.debug("{} ", args);

// Get the values of all parameters and concatenate all of them
String valueToAdd = "";
for(i=0;i<args.size();i++) {
    _logger.debug("Argument {}", i);
    Parameter thisParameter = args.get(i);
    _logger.debug("{} ", thisParameter);
    thisName = thisParameter.getName();
    thisValue = thisParameter.getValue();
    // Replace string separators
    thisValue = thisValue.replaceAll(" ", "%20");
    // Extend the new string value
    valueToAdd = valueToAdd + thisName + ":" + thisValue + "&";
}
if(i > 0) {
    if(valueToAdd.length() > 5) valueToAdd = valueToAdd.substring(0, valueToAdd.length() - 1);
    _logger.debug("Argument {} being added", i);
    // Add a new parameter with label and key 'combinedValue' to the list of parameters
    // of the operation and assign to it the previously created value
    args.addParameter(new Parameter("combinedValue", "combinedValue", valueToAdd));
    _logger.debug("{} ", valueToAdd);
    // Update transaction request with the new set of arguments
    lisa_vse_request.setArguments(args);

    _logger.debug("All Arguments now:");
    _logger.debug("{} ", lisa_vse_request.getArguments());
}
}
```

This script logs following information for a LisaBank transaction:

```
DEBUG com.itko.lisa.script.logger - POST /lisabank/buttonclick.do
DEBUG com.itko.lisa.script.logger - All Arguments:
DEBUG com.itko.lisa.script.logger - accountid=7740956037&userid=lisa_simpson&action=Withdraw
DEBUG com.itko.lisa.script.logger - Argument 0
DEBUG com.itko.lisa.script.logger - accountid=7740956037
DEBUG com.itko.lisa.script.logger - Argument 1
DEBUG com.itko.lisa.script.logger - userid=lisa_simpson
DEBUG com.itko.lisa.script.logger - Argument 2
DEBUG com.itko.lisa.script.logger - action=Withdraw
DEBUG com.itko.lisa.script.logger - Argument 3 being added
DEBUG com.itko.lisa.script.logger - accountid:7740956037&userid:lisa_simpson&action:Withdraw
DEBUG com.itko.lisa.script.logger - All Arguments now:
DEBUG com.itko.lisa.script.logger -
accountid=7740956037&userid=lisa_simpson&action=Withdraw&combinedValue=accountid:7740956037&userid:lisa_simpson&action:Withdraw
```

Scripted Dataset

With DevTest 8.0.2 the use of ‘Scripted DataSet’ is supported.

The Scripted Dataset enables creation and usage of test data by a script. The script maintains the current 'state' of the dataset across calls using SharedModelMap or PersistentModelMap. It passes the map to the test step for the test step to retrieve the value from the map.

DevTest Documentation

- [1] - DevTest Solutions: Using the Workstation and Console with CA Application Test – [Data Sets](#)

Input Parameters

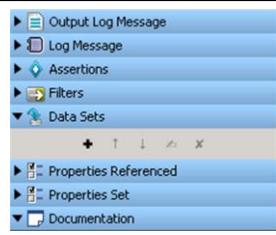
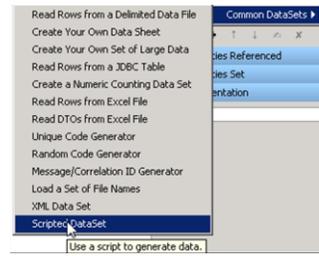
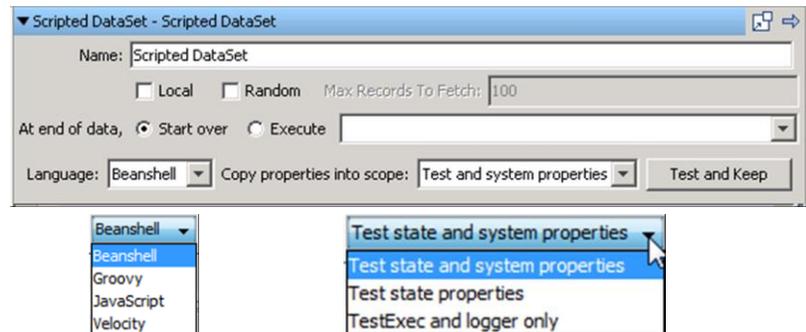
Other than the selected injected variables and properties the script does not have any specific input variables. Input data are retrieved from properties, usually.

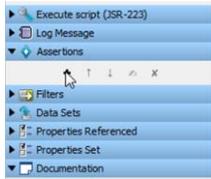
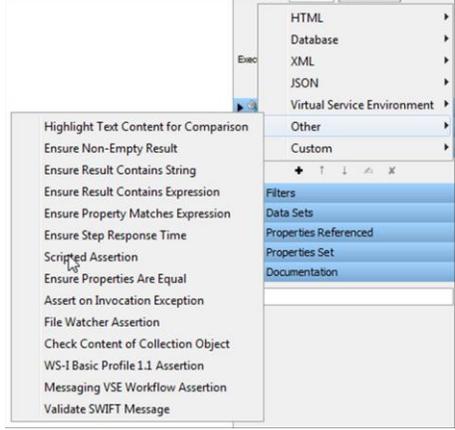
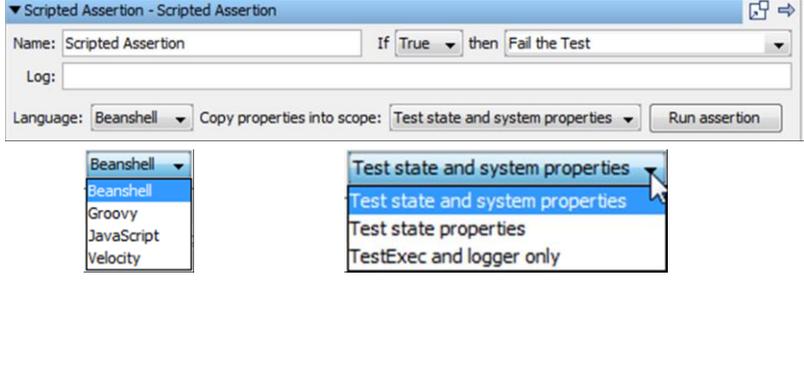
Output Parameters

The script must return a SharedModelMap or PersistentModelMap map variable.

Editor

The script editor for a scripted dataset opens in context of a test step.

	<p>In the Test case, select a test step, expand the ‘Data Sets’ node and click the ‘+’-sign to open the context menus for the various data sets available out of the box.</p>
	<p>From context menu select ‘Common DataSets > Scripted DataSet’, which opens the editor</p>
	<ul style="list-style-type: none"> • Name – specifies the name of the data set • On the use of ‘Local’ and ‘Random’ please see product documentation on data sets [1]. • On settings for ‘Start over’ and ‘Execute’ please see product documentation on data sets [1]. • Language – see Configuration Area • ‘Copy properties into scope’ – see Configuration Area and Object Selector

	<p>In the Test case, select a test step, expand the 'Assertions' on LHP and click the '+'-sign to open the context menus for the various assertions available out of the box.</p>
	<p>From context menu select 'Other > Scripted Assertion', which opens the editor</p>
	<ul style="list-style-type: none"> • Name – specifies the name of the assertion occurring in the list of assertions • 'If' clause specifies the condition of the script's return value when the assertion will trigger • 'Then' defines the action to execute when the condition is met • Log – specifies the event text to print to the event log when the assertion triggers • Language – see Configuration Area 'Copy properties into scope' – see Configuration Area and Object Selector • 'Run Assertion' – to open a window with the result of the script execution or a description of the errors that occurred.

Sample

This sample shows how to maintain the current status of a scripted data set in a Shared Model Map and how to pass the data to the test step. This Scripted Data Set sample is part of test case 'scriptedDataSet.tst', which itself is part of the examples project in DevTest 8.0.2

This is an example data set which is able to save state across executions within the same test. This is a common use case for custom data sets that read proprietary file formats, for example. All this sample really does is to count from 1 to 10 and pass the current value by the Shared Model Map to the test case. This approach can be used to save the current file position or cursor data or the last primary key value used. The 'state' of the dataset is saved across calls using SharedModelMap or PersistentModelMap. Both of them save and retrieve String values only so we need to do some data conversion.

<pre> // is there already a value from a previous call to the dataset? var currentValue = com.itko.lisa.vse.SharedModelMap.get("myNamespace", "myValue") if (currentValue === null) currentValue = "0" </pre>	<p>This Javascript sample is quite straight forward. Initially it checks if a Shared Model Map 'myNameSpace' exists and contains a value for property 'myValue'. If it does not variable 'currentValue' is initialized to 0.</p>
---	--

<pre>// increment var newValue = (Number(currentValue) + 1).toString() // save com.itko.lisa.vse.SharedModelMap.put("myNamespace", "myValue", newValue)</pre>	<p>The next two steps increment and then save the current value in the Shared Model Map.</p>
<pre>// return var map = null if (Number(newValue) >= 10) { _logger.info("Dataset has reached the end. clearing saved state and returning null") com.itko.lisa.vse.SharedModelMap.remove("myNamespace", "myValue") } else { map = new java.util.HashMap() map.put("myValue", newValue) map.put("someOtherValue", "cheese") }</pre>	<p>Now the Shared Model Map is prepared for passing to the test step. If the limit is reached the current Shared Model Map is cleared. Otherwise the new value is stored in the map.</p>
<pre>// javascript doesn't have explicit return so implicitly // return the last evaluated expression, in this case // our data map which contains two values... map</pre>	<p>In Javascript there is no explicit return value, but the last evaluated expression is returned.</p>

Appendix

The Appendix section contains information that does not fit yet into other chapters or sections of this document.

Performance considerations

- Some scripts can be compiled, others need interpretation
- inline `{{=%javascript% doSomething()}}` style scripts are NEVER compiled
- setup cost (variable injection)
- Some actual performance numbers
 - On a 2014 Mac Book Pro 2.3Ghz i7,
 - 'Do nothing' LISA step - 225k steps/second
 - Trivial groovy scripting step - 90k steps/sec
 - Trivial JavaScript step - 62k steps/sec
 - Legacy BeanShell scripting step - 38k steps/sec

Sample Code

SampleCode.zip contains files with sample code used in the document:

File name	Comment
Logger-Sample-1	Sample <code>_logger</code> statements representing different log levels
Logger-Sample-2	Sample <code>_logger</code> statements for different object types
Logger-Sample-3	Sample <code>_logger</code> statement to log complex objects
testExec-raiseEvent-Sample-1	Sample <code>testExec.raiseEvent()</code> statement for custom events
EmbdExpr-Sample-1	Samples for embedded expressions in different scripting languages used in test step 'Output Log Message'. This sample is taken from test case 'scripting-1.tst', which is part of the examples project in DevTest 8.

TestStep-Sample-1	Sample BeanShell script used in test step 'Execute Script (JSR-223)'. This script returns a 'work day' (<> Sat, Sun) with an offset of days to a given start date.
VSE-Router-Sample-1	Sample script returns a VSE execution mode based on a property if virtual service is running in DYNAMIC mode
MS-Sample-1	Default Match Script sample, which is available out of the box upon right mouse click. It demonstrates the usage of injected variables 'incomingRequest' and 'sourceRequest', and of invocation of the default matching logic.
MS-Sample-2	Match Script sample that demonstrates specific VSE logging capability in addition to usage of the injected variables and default matching logic
DPH-Sample-1-req	This Scriptable DPH sample showcases the usage of ShardeModelMap in the request part of a DPH, injected variable 'lisa_vse_request' and usage of classes 'ParameterList' and 'Parameter'
DPH-Sample-1-recrsp	This Scriptable DPH sample showcases the usage of SharedModelMap in the response part of a DPH, injected variable 'lisa_vse_response' and usage of class 'ParameterList'.
DPH-Sample-2-req	This DPH sample demonstrates how to add a single parameter to an operation using 'addParameter()' correctly.
DPH-Default-req	This is the demo DPH that is inserted out of the box if a Scriptable DPH was selected for the transaction request. It gives an overview on available methods for injected variable 'lisa_vse_request' and classes 'ParameterList' and 'Parameter'.
DPH-Default-recrsp	This is the demo DPH that is inserted out of the box if a Scriptable DPH was selected for one of the transaction responses. It gives an overview on available methods for injected variable 'lisa_vse_response' and classes 'ParameterList' and 'Parameter'.