Custom Sending Via Java

CloudTest version 2
Introduction 2
Custom Modules 2
Implementing custom “Messages” using Java 2
Example 3

CloudTest version

This document applies to CloudTest build 6592 and later.

Introduction

This document contains an example of how to use custom Java code to implement a custom "Message" in CloudTest. Your license must have "Custom Modules" enabled in order to do this.

You can use custom Java code to take the place of a "Message" in a Clip. Although typically such Java code would do some sort of sending and receiving, perhaps using a protocol and/or communication method not currently supported by CloudTest, the Java code can do anything that Java can do – it doesn't necessarily have to "send" or "receive".

The Java operations can be represented in the Result as if they were standard Messages in the Clip and can be included in the statistics as if they were a standard Message. Java operations can be mixed in with regular Messages (or any other standard item) in a Clip.

Custom Modules

A "Custom Module" is a Java jar that has been imported as an object into the CloudTest Repository. Once the Custom Module is created, it can be attached to Scripts (in the CloudTest Script editor) and then the JavaScript inside those Scripts can create Java objects that reside in the attached Custom Modules and call methods on those Java objects.

The Java code in Custom Modules can be used for any purpose, for example to do custom validations, custom formatting of data to be sent, custom parsing of data received, or any computation or processing that either cannot be done in JavaScript, is better done in Java, or for which some useful standard Java library already exists.

Note that security is applied that prevents the Java code from reading or writing to the server's disk, modifying Java security properties, and things of that nature.

Implementing custom "Messages" using Java

In this document, we're going to discuss using Java code in Custom Modules specifically for the purpose of implementing a custom "Message".

These custom "Messages" do not reside as separate objects in the Clip – they are implemented by code within CloudTest Scripts.

There are two distinct and independent parts to implementing a custom "Message".

First, as discussed above, Custom Modules can be used invoke Java code for any purpose. In this case, the Java code would be invoked to do whatever operations are required to implement the "Message" (typically some sort of "sending" and "receiving"). Note that it is not required that a single Java call be made – the Script could make a series of Java calls in order to implement the operation.

After the Java code has been executed, although the operation itself has been performed, CloudTest at this point has no knowledge that a custom "Message" has been executed.

The second part of the procedure involves informing CloudTest of the "Message", what was sent and received, and the statistics. This is done by calling the "`recordActionCompletion`" method on the "Result" object. This causes the information to be recorded into the current Result as if a regular Message had been executed.

The "`recordActionCompletion`" method is described here:

http://cdn.soasta.com/productresource/download/jsDoc/latest/symbols/Result.html

Executing the Java code to perform the operation and recording the result of the operation are two independent operations performed by the Script. A Script can record as many simulated custom "Messages" as it likes, even if they don't necessarily correspond to any actual sending and receiving operations (although it is not clear when that would be useful!).

Typically, a Script would execute the Java code to implement the operation, which would return information about what happened to the Script (including statistics), and then the Script would record that information into the Result.

A Script is not limited to only one custom "Message"; it can do as many operations as it needs to. Therefore, a single Script could generate many custom "Messages".

Example

The following example shows the basic outline of implementing a custom "Message" via Java.

First, we need some Java code to implement the actual operation. In this example there are two Java classes:

- The "`ExampleSender`" class.

  This class has a single method, "`sendSomething`", that implements the send/receive operation. In our example, we just have a "`Thread.sleep()`" call as a placeholder for where the actual send/receive operation takes place. Naturally what happens there is implementation-specific.

  The input is a String to be sent, and the output is an "`ExampleSenderOutput`" object that contains all of the information to be returned to the calling Script. Naturally the inputs that are actually needed will be implementation-specific.

- The "`ExampleSenderOutput`" class.

  This class contains the information that is returned to the calling Script.

The above classes are compiled into a Java jar, which is then imported into the CloudTest Repository as a Custom Module named "ExampleSender". You may use any external tool you desire to compile the Java code and create the jar.

The "`ExampleSender`" class:

```
package org.example;

import org.example.ExampleSenderOutput;

public class ExampleSender
{
  public ExampleSender()
  {
  }

  public ExampleSenderOutput sendSomething(String textToSend) throws Throwable
  {
    // Insert code to do the actual sending here.
    Thread.sleep(3000);

    ExampleSenderOutput out = new ExampleSenderOutput();

    out.m_operation = "MyOperation";
    out.m_url = "http://somewhere.org/something";
    out.m_completionType = "Success";
    out.m_failureText = null;
    out.m_validationPassed = true;
    out.m_textSent = textToSend;
    out.m_textReceived = "Received text here";

    out.m_deltaFromScheduledStart = 0;
    out.m_duration = 1234;
    out.m_bytesSent = textToSend.length();
    out.m_bytesReceived = 321;
    out.m_totalPortRetryCount = 0;
    out.m_totalPortRetryTime = 0;
    out.m_connectionEstablishTime = 100;
    out.m_sendTime = 777;
    out.m_receiveTime = 555;
    out.m_totalTime = 3000;
    out.m_ttfb = 5;
    out.m_ttlb = 2000;
    out.m_dnsLookupDuration = 2;
    out.m_waitForConnectionPoolTime = 3;

    return out;
  }
}
```

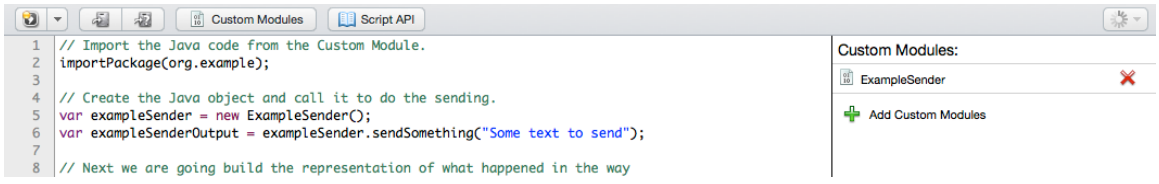
The "`ExampleSenderOutput`" class:

```
package org.example;

public class ExampleSenderOutput
{
  public String m_operation;
  public String m_url;
  public String m_completionType;
  public String m_failureText;
  public boolean m_validationPassed;
  public String m_textSent;
  public String m_textReceived;

  public long m_deltaFromScheduledStart;
  public long m_duration;
  public long m_bytesSent;
  public long m_bytesReceived;
  public long m_totalPortRetryCount;
  public long m_totalPortRetryTime;
  public long m_connectionEstablishTime;
  public long m_sendTime;
  public long m_receiveTime;
  public long m_totalTime;
  public long m_ttfb;
  public long m_ttlb;
  public long m_dnsLookupDuration;
  public long m_waitForConnectionPoolTime;
}
```

Next, we'll create the CloudTest Script that calls the Java code and records what happened into the Result. We'll call this Script "CustomModuleExampleScript":

```
// Import the Java code from the Custom Module.
importPackage(org.example);

// Create the Java object and call it to do the sending.
var exampleSender = new ExampleSender();
var exampleSenderOutput = exampleSender.sendSomething("Some text to send");

// Next we are going build the representation of what happened in the way
// that we want it to appear in the Result.

// Create a simulated Target.
var pretendTarget = new Object();
pretendTarget.name = "My Target";
pretendTarget.repositoryPath = "/Some folder";
pretendTarget.repositoryName = "My pretend Repository Target";

// Create a simulated "action" (Message), using the information and
// statistics returned from the Java call.
var pretendAction = new Object();
pretendAction.name = "My action";
pretendAction.repeatIndex = -1;
pretendAction.operation = exampleSenderOutput.m_operation;
pretendAction.url = exampleSenderOutput.m_url;
pretendAction.failureText = exampleSenderOutput.m_failureText;
pretendAction.validationPassed = exampleSenderOutput.m_validationPasses;
pretendAction.textSent = exampleSenderOutput.m_textSent;
pretendAction.textReceived = exampleSenderOutput.m_textReceived;
pretendAction.duration = exampleSenderOutput.m_duration;
pretendAction.completionType = exampleSenderOutput.m_completionType;
pretendAction.deltaFromScheduledStart = exampleSenderOutput.m_deltaFromScheduledStart;
pretendAction.bytesSent = exampleSenderOutput.m_bytesSent;
pretendAction.bytesReceived = exampleSenderOutput.m_bytesReceived;
pretendAction.totalPortRetryCount = exampleSenderOutput.m_totalPortRetryCount;
pretendAction.connectionEstablishTime = exampleSenderOutput.m_connectionEstablishTime;
pretendAction.sendTime = exampleSenderOutput.m_sendTime;
pretendAction.receiveTime = exampleSenderOutput.m_receiveTime;
pretendAction.totalTime = exampleSenderOutput.m_totalTime;
pretendAction.ttfb = exampleSenderOutput.m_ttfb;
pretendAction.ttlb = exampleSenderOutput.m_ttlb;
pretendAction.dnsLookupDuration = exampleSenderOutput.m_dnsLookupDuration;
pretendAction.waitForConnectionPoolTime = exampleSenderOutput.m_waitForConnectionPoolTime;

// Record the action in the Result.
$context.result.recordActionCompletion(pretendTarget, pretendAction);
```

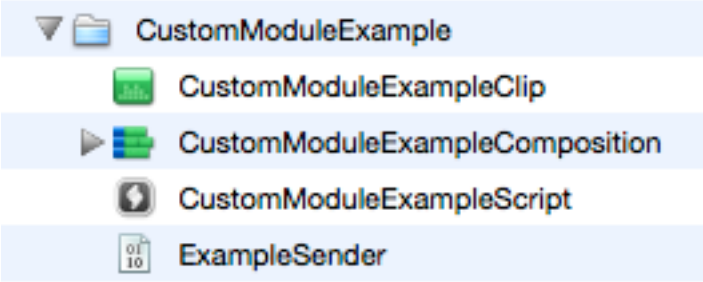
Note that this Script has the Custom Module attached:



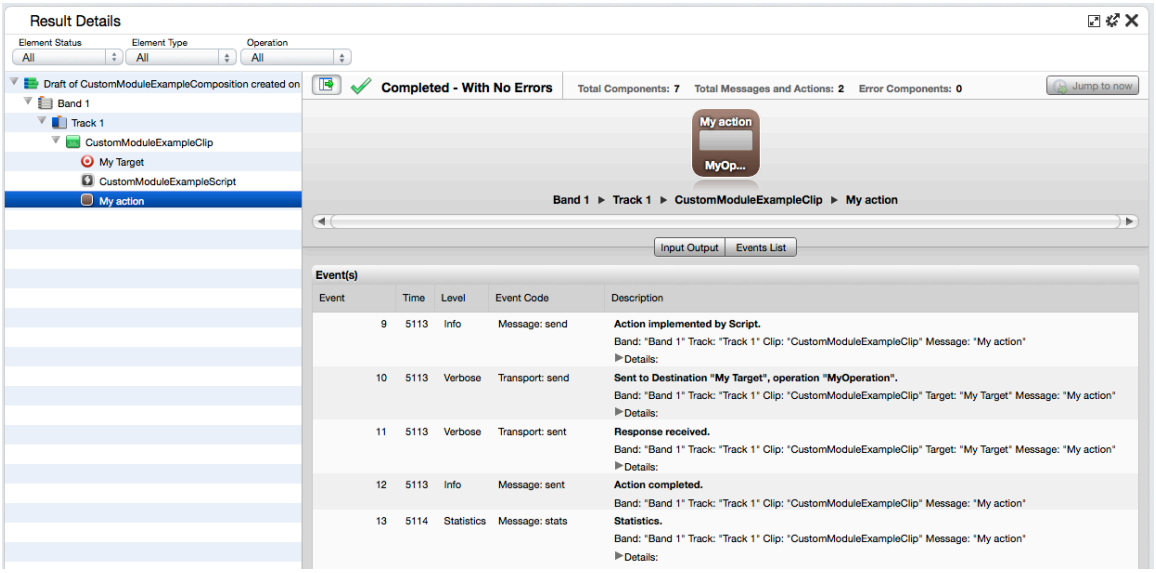
For a description of all of the information that can be passed to the "`recordActionCompletion`" method, refer to the link provided previously.

An alternative approach for this Script would be to have the "`sendSomething`" method return an object that exactly matches the members expected for the "action" parameter of the "`recordActionCompletion`" call. In that case, the returned object could be passed directly to the "`recordActionCompletion`" call without the need for JavaScript to transfer the values. Which approach is best will probably depend upon the individual situation.

Finally, we'll place the Script into a Clip called "CustomModuleExampleClip, and in turn place that Clip into a Composition named "CustomModuleExampleComposition". Thus, we end up with the following objects in CloudTest:



When the Composition plays, it produces the following Result:



Note that the action has appeared in the Result as if it were a Message, even though there was no actual Message object in the Clip, and the events for the Message reflect the information and statistics returned from the Java operation.

Time	Level	Event Code	Description
5113	Info	Message: send	**Action implemented by Script.** Band: "Band 1" Track: "Track 1" Clip: "CustomModuleExampleClip" Message: "My action" ▼Details: `Script "CustomModuleExampleScript", Clip "CustomModuleExampleClip", Track "Track 1", Band "Band 1"`
5113	Verbose	Transport: send	**Sent to Destination "My Target", operation "MyOperation".** Band: "Band 1" Track: "Track 1" Clip: "CustomModuleExampleClip" Target: "My Target" Message: "My action" ▼Details: `Some text to send`
5113	Verbose	Transport: sent	**Response received.** Band: "Band 1" Track: "Track 1" Clip: "CustomModuleExampleClip" Target: "My Target" Message: "My action" ▼Details: `Received text here`
5113	Info	Message: sent	**Action completed.** Band: "Band 1" Track: "Track 1" Clip: "CustomModuleExampleClip" Message: "My action"

5114	Statistics	Message: stats	**Statistics.**
			Band: "Band 1" Track: "Track 1" Clip: "CustomModuleExampleClip" Message: "My action"
			▼Details:
			`Sent at time: 3,871`
			`Delta from scheduled send time: 0 ms`
			`Total time: 3,000 ms`
			`Response time: 1,234 ms`
			`Time to establish connection: 100 ms`
			`Send time: 777 ms`
			`Receive time: 555 ms`
			`Time to first byte: 5 ms`
			`Time to last byte: 2,000 ms`
			`DNS lookup time: 2 ms`
			`Connection pool wait time: 3 ms`
			
			`Bytes sent: 17`
			`Bytes received: 321`
			
			`Retries due to unavailable local ports: 0`
			`Retry time due to unavailable local ports: 0 ms`