

A Step-By-Step Script Tutorial

Table of contents

1 Introduction.....	2
2 Script Imports.....	2
3 Test Definition.....	2
4 Bread crumbs.....	2
5 The Test Interface.....	3
6 Using the Dictionary and Random Python Modules.....	3
7 Forget the Java IO Package when Handling Files.....	3
8 Sending the Request and the Statistics API.....	4
9 Full Script Listing.....	4

1. Introduction

This is a step-by-step tutorial of how to write a number of dynamic HTTP tests using various aspects of The Grinder and Jython APIs. The test script contains a number of tests that are requests to the same URL. For each request, a different XML parameter is specified. The resulting HTML data is checked on return and if the test was not successful, the statistics API is used to mark that test as failed.

Richard Perks

2. Script Imports

```
import string
import random
from java.lang import String
from java.net import URLDecoder
from net.grinder.script import Test
from net.grinder.plugin.http import HTTPRequest
from net.grinder.common import GrinderException
```

Firstly when writing a script come the import statements. These include imports of standard Python modules such as `string` and `random`, and other Java imports including some language and network classes. Finally there are imports for Grinder specific methods. A powerful feature of the Jython scripts that are used with The Grinder is the ability to take a mix and match approach to script programming. In some cases using a Python API is quicker and easier than always using the corresponding Java API calls, so feel free to use whichever API makes most sense.

3. Test Definition

```
tests = {
    "News01"       : Test(1, "News 1 posting"),
    "Sport01"      : Test(2, "Sport 1 posting"),
    "Sport02"      : Test(3, "Sport 2 posting"),
    "Trading01"    : Test(4, "Trading 1 query"),
    "LifeStyle01"  : Test(5, "LifeStyle 1 posting"),
}
```

To keep the script code easy to read, we next define all the tests we are going to be running within this script. These are created as a Python dictionary and are name-value pairs. The name is the name of the test and the value is a `Test` object with a test numeric identifier and description.

4. Bread crumbs

```
log = grinder.logger.output
out = grinder.logger.TERMINAL

# Server Properties
SERVER    = "http://serverhost:7001"
URI       = "/myServlet"
```

We next define some variables such as Grinder helper methods and server properties. The

`log` variable is used to hold a reference to The Grinder logging mechanism and is used throughout the script. The `out` variable is also used in conjunction with logging. The two possible values for this variable are `TERMINAL` and `LOG`. We have set the output to terminal for ease of debugging, which means that any log output goes to the terminal window used to start The Grinder test runs. The alternative is to switch this to logging the output to The Grinder log files.

5. The Test Interface

```
class TestRunner:
    def __call__(self):
```

Here is the definition of our test class and the method called by The Grinder by each test thread. All scripts must define this class and method. Whilst we are discussing classes and methods, an important point to remember when new to Jython script development is that Jython/Python code is scoped by indentation, rather than using braces like in a language like C or Java. The colon is used to delimit the start scope such as an `if` or method definition.

6. Using the Dictionary and Random Python Modules

```
for idx in range(len(tests)):
    testId = random.choice(tests.keys())
    log("Reading XML file %s " % testId, out)
```

As discussed earlier, the use of Python modules is encouraged during Grinder script development and I have used a few examples above when performing the test run. Within the test run, each of the tests defined in the test dictionary is looped round so that each Grinder thread executes five separate tests. Within the loop, a test is chosen randomly from one of the five tests. This prevents all threads of executing all the tests in the same order and helps simulate a more random load on the server.

Within the dictionary defined as `tests`, there are a number of useful methods such as `keys()`, `items()` and `sort()`. We use the keys returned from the tests dictionary as the parameter to the `choice()` method in the random module. This randomly selects one of the tests keys as the current test identifier.

7. Forget the Java IO Package when Handling Files

```
file = open("./CAAssets/"+testId+".xml", 'r')
fileStr = URLEncoder.encode(String(file.read()))
file.close()
```

```
requestString = "%s%s%s%s" % (SERVER, URI, "?xmldata=", fileStr)
```

When having to retrieve the contents of files using Jython script, the use of the file operations blitz's Java IO for pure script development speed. In the code above, we need to open an XML document that has the name of a test, for example `News01.xml`. This will be used as a request parameter for the `News01` test. The file is opened for reading and encoded using the Java `URLEncoder`.

We next construct the request string to the server by concatenating the server, URI and

XML documents together. *Tip:* if you need to remove spaces from within a string, you can use a method like the following:

```
requestString = string.join(requestString.split(), "")
```

8. Sending the Request and the Statistics API

```
grinder.statistics.delayReports = 1
request = tests[testId].wrap(HTTPRequest())

log("Sending request %s " % requestString, out)
result = request.GET(requestString)
```

As part of the test execution, we want the ability to check the result of the HTTP request. If the response back from the server is not one that we expect, we want to mark the test as unsuccessful and not include the statistics in the test times. To do this, the `delayReports` variable can be set to 1. Doing so will delay the reporting back of the statistics until after the test has completed and we have had chance to check its operation. The default is to report back when the test returns control back to the script, i.e. immediately after a test has executed.

Next we wrap the `HTTPRequest` with the test being executed. This enables any calls through the test wrapper to be monitored by the Grinder. Only wrapped tests will be used when collecting test statistics. Any other time spent within the script will not be recorded by The Grinder. Be careful not to include extra script processing within a test; doing so will not give the correct statistics. Only test what is required.

The test itself is next executed which is a HTTP GET to the server using our previously constructed test string. Remember - these tests execute in a loop for the number of tests we have defined, using a random test each time.

```
if string.find(result.getText(), "SUCCESS") < 1:
    grinder.statistics.forLastTest.setSuccess(0)
    writeToFile(result.getText(), testId)
```

On return from the HTTP GET, we check the result for the string "SUCCESS". If the test has failed, this value will not be returned and the statistics object can be marked as unsuccessful. In the case of an unsuccessful test, we write the HTML output to a file for later analysis:

```
def writeToFile(text, testId):
    filename = grinder.getFilenameFactory().createFilename(
        testId + "-page", "-%d.html" % grinder.runNumber)

    file = open(filename, "w")
    print >> file, text
    file.close()
```

9. Full Script Listing

```
# Send an HTTP request to the server with XML request values

import string
import random
from java.lang import String
from java.net import URLEncoder
```

```
from net.grinder.script import Test
from net.grinder.plugin.http import HTTPRequest
from net.grinder.common import GrinderException

tests = {
    "News01"      : Test(1, "News 1 posting"),
    "Sport01"     : Test(2, "Sport 1 posting"),
    "Sport02"     : Test(3, "Sport 2 posting"),
    "Trading01"  : Test(4, "Trading 1 query"),
    "LifeStyle01": Test(5, "LifeStyle 1 posting"),
}

log = grinder.logger.output
out = grinder.logger.TERMINAL

# Server Properties

SERVER      = "http://serverhost:7001"
URI         = "/myServlet"

class TestRunner:
    def __call__(self):
        for idx in range(len(tests)):
            testId = random.choice(tests.keys())

            log("Reading XML file %s " % testId, out)

            file = open("./CAAssets/"+testId+".xml", 'r')
            fileStr = URLEncoder.encode(String(file.read()))
            file.close()

            # Send the request to the server
            requestString = "%s%s%s%s" % (SERVER, URI, "?xmldata=",
fileStr)
            requestString = string.join(requestString.split(), "")

            grinder.statistics.delayReports = 1
            request = tests[testId].wrap(HTTPRequest())

            log("Sending request %s " % requestString, out)
            result = request.GET(requestString)

            if string.find(result.getText(), "SUCCESS") < 1:
                grinder.statistics.forLastTest.setSuccess(0)
                writeFile(result.getText(), testId)

# Write the response
def writeFile(text, testId):
    filename = grinder.getFilenameFactory().createFilename(
        testId + "-page", "-%d.html" % grinder.runNumber)

    file = open(filename, "w")
    print >> file, text
    file.close()
```