Scripts

Table of contents

This section describes The Grinder 3 scripting API. If you've used The Grinder 2 for HTTP testing and you're not a programmer, you might be a bit daunted. Don't worry, its just as easy to record and replay HTTP scripts with The Grinder 3.

1 Jython and Python

The default scripting engine is Jython - the Java implementation of Python. Python is powerful, popular and easy on the eye. If you've not seen any Python before, take a look at the script gallery (../g3/script-gallery.html) and Richard Perks' tutorial (../g3/tutorial-perks.html) to get a taste of what its like. There are plenty of resources on the web, here are a few of them to get you started:

- The Jython home page (http://www.jython.org/)
- The Python language web site (http://www.python.org/)
- Ten Python pitfalls (http://zephyrfalcon.org/labs/python_pitfalls.html)

I recommend the Jython Essentials (http://www.amazon.com/exec/obidos/tg/detail/-/0596002475/qid%3D1044795121/103-7145719-3118225) book; you can read the introductory chapter (http://www.oreilly.com/catalog/jythoness/chapter/ch01.html) for free.

1.1 Alternative languages

The Grinder 3.6 and later support test scripts written in Clojure (../g3/tcpproxy.html#clojure-script) .

Ryan Gardner has written an add-on script engine for Groovy (http://code.google.com/p/grinder-maven-plugin) .

2 Jython scripting

2.1 Script structure

Jython scripts must conform to a few conventions in order to work with The Grinder framework. I'll lay the rules out in fairly dry terms before proceeding with an example. Don't worry if this makes no sense to you at first, the examples are much easier to comprehend.

1. **Scripts must define a class called `TestRunner`**

   When a worker process starts up it runs the test script once. The test script must define a class called `TestRunner`. The Grinder engine then creates an instance of `TestRunner` for each worker thread. A thread's TestRunner instance can be used to store information specific to that thread.

   > **Note:**
   > Although recommended, strictly `TestRunner` doesn't need to be a class. See the Hello World with Functions (../g3/script-gallery.html#helloworldfunctions.py) example.

2. **The `TestRunner` instance must be callable**

   A Python object is callable if it defines a `__call__` method. Each worker thread performs a number of *runs* of the test script, as configured by the property `grinder.runs`. For each run, the worker thread calls its `TestRunner`; thus the `__call__` method can be thought of as the definition of a run.

3. **The test script can access services through the `grinder` object**

   The engine makes an object called `grinder` available for the script to import. It can also be imported by any modules that the script calls. This is an instance of the Grinder.ScriptContext (../../g3/script-javadoc/net/grinder/script/Grinder.ScriptContext.html) class and provides access to context information (such as the worker thread ID) and services (such as logging and statistics).
4. **The script file name must end in `.py`**

   The file name suffix is used to identify Jython scripts.

2.2 Canonical test script structure

This is an example of a script that conforms to the rules above. It doesn't do very much - every run will log *Hello World* to the worker process log.

```
from net.grinder.script.Grinder import grinder

# An instance of this class is created for every thread.
class TestRunner:
    # This method is called for every run.
    def __call__(self):
        # Per thread scripting goes here.
        grinder.logger.info("Hello World")
```

2.3 Automatically generating scripts

If you are creating a script for a website or web application, you can use the TCPProxy (../g3/tcpproxy.html#HTTPPluginTCPProxyFilter) to generate an HTTPPlugin script suitable for use with The Grinder.

3 Tests

Although our simple test script can be used with The Grinder framework and can easily be started in many times in many worker processes on many machines, it doesn't report any statistics. For this we need to create some tests. A Test (../../g3/script-javadoc/net/grinder/script/Test.html) has a unique test number and description. If you are using the console (../g2/console.html) , it will update automatically to display new `Test`s as they are created.

Let's add a `Test` to our script.

```
from net.grinder.script import Test
from net.grinder.script.Grinder import grinder

# Create a Test with a test number and a description.
test1 = Test(1, "Log method")

class TestRunner:
    def __call__(self):
        grinder.logger.info("Hello World")
```

Here we have created a single `Test` with the test number *1* and the description *Log method*. Note how we import the `grinder` object and the `Test` class in a similar manner to Java.

Now the console knows about our `Test`, but we're still not using it to record anything. Let's record how long our `grinder.logger.info` method takes to execute.

`Test.record` adds the appropriate instrumentation code to the byte code of method. The time taken and the number of calls will be recorded and reported to the console.

```
from net.grinder.script import Test
from net.grinder.script.Grinder import grinder

test1 = Test(1, "Log method")

# Instrument the info() method with our Test.
test1.record(grinder.logger.info)

class TestRunner:
    def __call__(self):
        grinder.logger.info("Hello World")
```

This is a complete test script that works within The Grinder framework and reports results to the console.

You're not restricted to instrument method calls. In fact, it's more common to instrument objects. Here's an example using The Grinder's HTTP plug-in (../g3/http-plugin.html) .

```
# A simple example using the HTTP plugin that shows the retrieval of a
# single page via HTTP.

from net.grinder.script import Test
from net.grinder.script.Grinder import grinder
from net.grinder.plugin.http import HTTPRequest

test1 = Test(1, "Request resource")
request1 = HTTPRequest()
test1.record(request1)

class TestRunner:
    def __call__(self):
        result = request1.GET("http://localhost:7001/")
```

4 The Grinder script API

With what you've seen already you have the full power of Jython at your finger tips. You can use practically *any* Java or Python code in your test scripts.

The Grinder script API can be used to access services from The Grinder. The Javadoc (../../g3/script-javadoc/index.html) contains full information on all the packages, classes and interfaces that make up the core API, as well as additional packages added by the shipped plug-ins. This section provides overview information on various areas of the API. See also the HTTP plugin documentation (../g3/http-plugin.html) .

The net.grinder.script (../../g3/script-javadoc/net/grinder/script/package-summary.html) package

An instance of Grinder.ScriptContext (../../g3/script-javadoc/net/grinder/script/Grinder.ScriptContext.html) called `grinder` is automatically available to all scripts. This object provides access to context information and acts a starting point for accessing other services. The instance can be explicitly imported from other Python modules as `net.grinder.script.Grinder.grinder`.

We have described the use of the Test (../../g3/script-javadoc/net/grinder/script/Test.html) class above.

The Statistics (.././g3/script-javadoc/net/grinder/script/Statistics.html) interface allows scripts to query and modify statistics (../g3/statistics.html) , provide custom statistics, and register additional views of standard and custom statistics.

The net.grinder.common (.././g3/script-javadoc/net/grinder/common/package-summary.html) package

This package contains common interfaces and utility classes that are used throughout The Grinder and that are also useful to scripts.

5 Working directory

When the script has been distributed using the console, the working directory (CWD) of the worker process will be the local agent's cache of the distributed files. This allows the script to conveniently refer to other distributed files using relative paths.

Otherwise, the working directory of the worker process will be that of the agent process that started it.

5.1 Distributing Java code

You can add Java `jar` or `.class` files to your console distribution directory and use the file distribution mechanism to push the code to the agent's cache. Use relative paths and the `grinder.jvm.classpath` property to add the files to the worker process `CLASSPATH`.

For example, you might distribute the following files

```
grinder.properties
myscript.py
lib/myfile.jar
```

where `grinder.properties` contains:

```
grinder.script=myscript.py
grinder.jvm.classpath=lib/myfile.jar
```