Test Automation Rules Engine

Programming Guide

The *Test Automation Rules Engine* is a tool that can be used for regression or features testing, system configuration or production rollout validation. It can be used like a scripting language but it has increased validation capacities. These validation features are what make the Rules Engine so powerful for testing and validation.

There are actually two main components. One is the *Rules Engine* and the other is a *Reporting Interface* which can be thought of as a wrapper for the Rules Engine. The Reporting Interface takes commands and information from Jira tickets and reports back the test results and exhibit attachments to Jira.

Table of Contents

Rules Engine Introduction

The Rules Engine is executed using *RulesNgn.py*. It also uses the module RulesLib.py. The Rules Engine takes instructions from a *Ruleset* input file. *Rulesets* are collections of *Rules*, each which take on the form of *Actions* and *Validations*. Actions can be system commands or specific functions as described below. Validations are a *Status Definition* followed by a *Test Condition*.

The Ruleset file is a CSV file, and can be edited with a text editor or a spreadsheet program. Any line that begins with a "#" (pound) character is treated as a comment. The "#" (pound) character also serves as a delimiter between actions and validations when it stands alone in a field.

Each line of the Ruleset file can have one or more actions separated by commas, and/or one or more validations separated by commas. The actions and validations are separated by a field containing the "#" (pound) character as a delimiter. The delimiter field can have one or more "#" (pound) characters, as desired for readability.

Line of the Ruleset file:

action1, action2, action3, etc, ##### , validation1, validation2, validation3, etc.

Lines are not required to have actions, nor are they required to have validations. It wouldn't make much sense to have a line that had neither an action nor a validation, but it is permissible. It is definitely useful to have lines with only an action or only a validation. However, the output of a command can only be tested on the line the command is executed on.

One useful case of an action without a validation is some system command that can be expected to run, and doesn't really need to be tested. In this case, no validation might be tested.

A useful case of a validation without an action is a log parser. The ruleset might launch a command which then generates a logfile or some other testable output. If multiple things should be validated, then they can be put on subsequent lines after the initial command. The only validation that *must* be put on the same line as the command is a validation that tests the (stdout or stderr) output of the command.

Sample Ruleset file:

```
# Rules.csv
#
# Sample Rules File

dir > dir.txt                          ,###,     PASS if exists dir.txt
touch none.txt                         ,###,     FAIL if missing none.txt
exit 0                                 ,###,     FAIL if error
exit 1                                 ,###,     PASS if error
exit 244                               ,###,     PASS if equal 244
exit 51                                ,###,     FAIL if not equal 51
exit 5                                 ,###,     PASS if has bits 4
exit 3                                 ,###,     FAIL if has bits 4
echo This is a test > test.txt         ,###,     FAIL if missing test.txt
                                       ,###,     PASS if contains "This" test.txt
echo stdout has data                   ,###,     PASS if stdout contains "has data"
echo stderr has errors 1>&2            ,###,     PASS if stderr contains "has errors"
echo stderr has data   1>&2            ,###,     PASS if output contains has data
echo 0000 0001 0002                    ,###,     PASS if stdout equal "0000 0001 0002"
echo 0000 0001 0002    1>&2            ,###,     PASS if stderr equal "0000 0001 0002"
echo 0000 0001 0002                    ,###,     PASS if stdout 0 equal 0000
echo 0000 0001 0002                    ,###,     PASS if stdout 1 equal 0001
echo 0000 0001 0002    1>&2            ,###,     PASS if stderr 2 equal 0002
echo 4                                 ,###,     PASS if stdout has bits 4
echo 4                                 ,###,     PASS if output has bits 4
echo 4                 1>&2            ,###,     PASS if stderr has bits 4
echo 1 2 4 8                           ,###,     PASS if stdout 1 has bits 2
echo 1 2 4 8           1>&2            ,###,     PASS if stderr 2 has bits 4
```

Supported Actions and Validations

Actions:

TOUCH <file>

WAIT ON FILE PRESENT <file>

WAIT ON FILE ABSENT <file>

WAIT <seconds>

CAPTURE <sourcefile> <number> **TO** <targetfile>

START <command> (runs command asynchronously, doesn't capture output or return code)

<**COMMAND**> (runs command synchronously, waits for output and return code)

TOUCH acts the same as the UNIX "touch" command. It updates the file timestamp of a file that exists, or creates a file of size zero if the file doesn't exist.

WAIT ON FILE PRESENT is pretty self-explanatory. Execution of the ruleset pauses until the specified file is present.

WAIT ON FILE ABSENT works the same way. Execution of the ruleset pauses until the specified file is NOT present.

WAIT followed by a number waits that number of seconds.

CAPTURE works like the "tail" command in UNIX. A targetfile is created containing the number of lines specified from the contents of the sourcefile. This creates a subset of the sourcefile that starts at the time the *capture* command is executed.

START launches a command line program or system command. Execution is asynchronous, e.g. run in the background.

COMMAND line programs or system commands can be launched as actions. Without the "START" prefix, the command is run synchronously, so the ruleset waits for the command to finish. The return code or (stdout or stderr) outputs of the command may be tested by the validation section of the Ruleset parser.

Validations:

Status Definitions:

PASS, **FAIL**, **WARN** or **EXIT**

Validation Conditions:

if equal / **if not equal** <value> (checks program return code)

if error (same as *if not equal 0*)

if has bits <value>

if exists <filename>

if missing <filename>

if contains <pattern> <filename>

if tail <filename> <number> **contains** <pattern>

if stdout [offset] **equal** <pattern> (looks at stdout string like args, offset optional, default 0)

if stdout [offset] **not equal** <pattern>

if stdout [offset] **has bits** <value>

if stdout contains <pattern> (examines entire stdout string)

if stderr [offset] **equal** <pattern> (looks at stderr string like args, offset optional, default 0)

if stderr [offset] **not equal** <pattern>

if stderr [offset] **has bits** <value>

if stderr contains <pattern> (examines entire stderr string)

if output [offset] **equal** <pattern> (looks at both stdout and stderr)

if output [offset] **not equal** <pattern>

if output [offset] **has bits** <value>

if output contains <pattern>

Status Definitions:

PASS provides a positive result when the validation condition is true.

FAIL provides a negative result when the validation condition is true.

WARN provides a positive result whether or not the validation condition is true, but gives a warning if the validation condition is true.

EXIT provides a negative result and terminates ruleset execution immediately if the validation condition is true.

Validation Conditions:

if equal <*value*> returns true if the return code of the action is equal to *value*.

if not equal <*value*> returns true if the return code is NOT equal to *value*.

if error returns true if the return code is NOT equal to *0*.

if has bits <*value*> returns true if the logical conjunction (Boolean AND) between *value* and the return code is true.

if exists <*filename*> returns true if *filename* is present.

if missing <*filename*> returns true if *filename* is absent.

if contains <*pattern*> <*filename*> returns true if *pattern* is found within *filename*.

if tail <*filename*> <*number*> contains <*pattern*> returns true if *pattern* is found within the last *number* lines in *filename.*

if stdout equal <*pattern*> returns true if *pattern* is equal to stdout from action command output. Equivalency is determined using a string comparison.

if stdout <*offset*> equal <*pattern*> returns true if *pattern* is equal to arg[*offset*] of stdout from action command output. So, for example, if the command returns "0000 0001 0002", then arg[0] is "0000", arg[1] is "0001" and arg[2] is "0002".

if stdout not equal <*pattern*> returns true if *pattern* is NOT equal to stdout from command.

if stdout <*offset*> not equal <*pattern*> returns true if *pattern* is NOT equal to arg[*offset*] of stdout from command.

if stdout has bits <*value*> returns true if the logical conjunction (Boolean AND) between *value* and stdout is true. The contents of stdout are expected to be a numeric value.

if stdout <*offset*> has bits <*value*> returns true if the logical conjunction (Boolean AND) between *value* and arg[*offset*] of stdout is true. The contents of stdout are expected to be a numeric value.

if stdout contains <*pattern*> returns true if stdout contains *pattern*.

if stderr equal <*pattern*> returns true if *pattern* is equal to stderr from action command output. Equivalency is determined using a string comparison.

if stderr <*offset*> equal <*pattern*> returns true if *pattern* is equal to arg[*offset*] of stderr from action command output. So, for example, if the command returns "0000 0001 0002", then arg[0] is "0000", arg[1] is "0001" and arg[2] is "0002".

if stderr not equal <*pattern*> returns true if *pattern* is NOT equal to stderr from command.

if stderr <*offset*> not equal <*pattern*> returns true if *pattern* is NOT equal to arg[*offset*] of stderr from command.

if stderr has bits <*value*> returns true if the logical conjunction (Boolean AND) between *value* and stderr is true. The contents of stderr are expected to be a numeric value.

if stderr <*offset*> has bits <*value*> returns true if the logical conjunction (Boolean AND) between *value* and arg[*offset*] of stderr is true. The contents of stderr are expected to be a numeric value.

if stderr contains <*pattern*> returns true if stderr contains *pattern*.

if output equal <*pattern*> returns true if *pattern* is equal to output (stdout+stderr) from action command output. Equivalency is determined using a string comparison.

if output <*offset*> equal <*pattern*> returns true if *pattern* is equal to arg[*offset*] of output from action command output. So, for example, if the command returns "0000 0001 0002", then arg[0] is "0000", arg[1] is "0001" and arg[2] is "0002".

if output not equal <*pattern*> returns true if *pattern* is NOT equal to command output.

if output <*offset*> not equal <*pattern*> returns true if *pattern* is NOT equal to arg[*offset*] of command output.

if output has bits <*value*> returns true if the logical conjunction (Boolean AND) between *value* and command output is true. The contents of output are expected to be a numeric value.

if output <*offset*> has bits <*value*> returns true if the logical conjunction (Boolean AND) between *value* and arg[*offset*] of command output is true. The contents of output are expected to be a numeric value.

if output contains <*pattern*> returns true if output contains *pattern*.

Reporting Interface Introduction

The Reporting Interface is a wrapper around the Rules Engine. It is executed using *TestX.py* and also uses the module ReprtLib.py. *TestX* takes a *manifest* file as its input, which includes metadata about the tests as well as test data and rulesets. If rulesets are included, then the Rules Engine is triggered and the resulting logfile is captured. The output is a JSON message with exhibits as attachments which can be digested by Jira for automated test execution and reporting.

The manifest file can be in one of two formats, raw text or JSON. The raw text format has key:value pairs separated with colons. The JSON format is exactly like the output format, and is useful as a potential pass-through mechanism when used with other systems' reporting mechanisms. A JSON manifest file that contains attachments from associated systems can be presented and these will be passed along to the output. If the manifest file contains a Ruleset, it will be run and evidence will be collected and attached along with the existing contents of the JSON manifest file.

There are a few differences between the formats beyond the formats, themselves. Said another way, the Reporting Interface interprets and processes the two files slightly differently.

For example, the raw text form of the manifest file has three fields for each record in the "evidences" section and five fields for each record in the "steps" section. Each field must be present, although the values can be blank.

In the JSON format, the evidences section must have three fields, none of which can be blank. The steps section must have "status" and "comments" fields populated or it must have "filename", "data" and "contentType" fields populated.

An implication of this is JSON attachments must be included as data within the JSON manifest file, whereas attachments can be placed on the local drive rather than embedded in the text form of the manifest file. The text version of the file can have a filename value but be absent the data, in which case the Reporting Engine will look for the file on the local drive.

Additionally, the JSON format manifest file supports a "rules" section. It is similar to the "steps" block, except it can only contain file structures using "filename", "data" and "contentType" fields. The rules block is not expected to have "status" or "comment" fields.

Sample Raw Text Manifest File:

Test Metadata

testKey : Jira-1000

status :

start :

finish :

comment : Test of the Rules Engine

examples :

defects :

evidences: data:filename:contentType

evidences :

data:TWFuIGlzIGRpc3Rpbmd1aXNoZWQsIG5vdCBvbmx5IGJ5IGhpcyByZWFzb24sIGJ1dCBieSB0aGlzIHNpbmd1bGFyIHBhc3Npb24gZnJvbSBvdGhlciBhbmltYWxzLCB3aGljaCBpcyBhIG1c3Qgb2YgdGhlIG1pbmQsIHRoYXQgYnkgYSBwZXJzZXZlcmFuY2Ugb2YgZGVsaWdodCBpbiB0aGUgY29udGludWVkIGFuZCBpbmRlZmF0aWdhYmxlIGdlbmVyYXRpb24gb2Yga25vd2xlZGdlLCBleGNlZWRzIHRoZSBzaG9ydCB2ZWhlbWVuY2Ugb2YgYW55IGNhcm5hbCBwbGVhc3VyZS4=,
filename:evidence.txt, contentType:text/plain

steps: status:comment:data:filename:contentType

steps : status:, comment:Rules File, data:, filename:rules.csv, contentType:text/csv

Info Metadata

summary : Test of the Rules Engine

description : Tests for both the Rules Engine and Reporting Interface

user : wparham

Sample JSON Manifest File:


```
{
	"info": {
		"summary": "Test of the Rules Engine and Reporting Interface",
		"description": "Test single ruleset using JSON manifest file",
		"user": "wparham"
	},
	"tests": [{
		"testKey": "Jira-1001",
		"comment": "Ruleset is embedded below",
		"steps": [{
			"filename": "rules.csv",
			"contentType": "application/csv",
			"data":
"IyBydWxlcy5jc3YNCiMNCiMgcHJpbnRzICJ0ZXN0IE9ORSIgYW5kIHZlcmlmaWVzIHRoZSBvdXRwd
XQNCg0KZWNobyB0ZXN0IE9ORSAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICBQQVNTIGlmIG91dHB1dCBlcXVhbCAid
GVzdCBPTkUi"
		}]
	}]
}
```

Sample Output File:


```
 {
   "info": {
     "summary": "Test of the Rules Engine and Reporting Interface",
     "description": "Test single ruleset using JSON manifest file",
     "user": "wparham",
     "startDate": "2019-02-28T12:37:37+0000",
     "finishDate": "2019-02-28T12:37:37+0000"
   },
   "tests": [{
     "testKey": "Jira-1001",
     "comment": "Ruleset is embedded below",
     "start": "2019-02-28T12:37:37+0000",
     "finish": "2019-02-28T12:37:37+0000",
     "status": "PASS",
     "evidences": [{
       "filename": "rules.log",
       "contentType": "text/plain",
       "data":
"PT0+IFByb2Nlc3NpbmcgWy4vcnVsZXMuY3N2XSBSdWxlcyBGaWxlLg0KPT0+IFJ1bGVzIEZpbGUg
Wy4vcnVsZXMuY3N2XSBjb250YWlucyAoNSkgbGluZXMsICgxKSBjb250YWluaW5nIHJ1bGVzLg0K
PT0+IFN0YXJ0IHJ1biBhdCAyMDE5LTAyLTI4IDEyOjM3OjM3DQogIDEgY21kOiBlY2hvIHRlc3QgT05
FDQogIDEgdHN0OiBQQVNTIGlmIG91dHB1dCBlcXVhbCAidGVzdCBPTkUiDQo9PT4gUEFTUyBhdC
AyMDE5LTAyLTI4IDEyOjM3OjM3DQo9PT4gUEFTUyBbLi9ydWxlcy5jc3ZdIC0gQWxsIHRlc3RzIHN1
Y2NlZWRlZCBhdCAyMDE5LTAyLTI4IDEyOjM3OjM3DQo="
       }
     ],
     "steps": [{
```

```
      "contentType": "application/csv",
      "filename": "rules.csv",
      "data":
"IyBydWxlcy5jc3YNCiMNCiMgcHJpbnRzICJ0ZXN0IE9ORSIgYW5kIHZlcmlmaWVzIHRoZSBvdXRwd
XQNCg0KZWNobyB0ZXN0IE9ORSAgICAgICAgICwjIyMsICAgICBQQVNTIGlmIG91dHB1dCBlcXVhbCAid
GVzdCBPTkUi"
      },
      {
      "status": "EXECUTING",
      "comment": "  1.  echo test ONE"
      },
      {
      "status": "PASS",
      "comment": "  1.  PASS if output equal \"test ONE\" at 2019-02-28 12:37:37"
      }
    ]
  }]
}
```

Additional Notes:

Any output file can be renamed “manifest” and re-run as an input file. This provides pass-through functionality to allow information to be concatenated by various systems. A single message can be passed from system to system, with information added to it as needed.

Once a manifest file is run, any Rulesets contained in the “steps” or “rules” sections will be run. The output logfile will be attached in the “evidences” section and a parsed version will be appended as status/comment values in the “steps” section.

Subsequent runs will not re-run a Ruleset, so the output file will exactly match the input file. But if new Rulesets are added, they will be run.

Files contained in the “evidences”, “steps” and “rules” sections are always decoded and written to the local drive for the duration of the run. They can be captured or kick off processes on the local machine. After TestX completes, the files written to the local drive are deleted but they are not deleted from the manifest file.