The ‘Toolbox’ Testing Approach

James Gibbard – Intel Security

james.gibbard@intel.com

Abstract

In an ever changing and increasing world of agile software development, automated testing approaches have struggled to keep up with the amount of software that can be developed. This has been exacerbated with the recent trend of using multiple development technologies to form a complex stack on which software products are created.

The usual approach to this problem from an automated test tooling perspective, is to make use of multiple, separate test tools/harnesses. These tools can only provide test automation for some of the complex development stack. This can be an expensive, heavyweight and inflexible solution, that struggles to provide testers the agility they need to implement robust, automated tests. Large COTS (Commercial Off The Shelf) software is normally rigid in its ability to be customized, or adapted to test different technologies it was not originally intended for.

This paper sets out a solution to the problem, with the use of a ‘toolbox’ test approach, leveraging the power of Robot Framework. The approach provides flexibility by allowing testers to utilize multiple automated testing techniques and technologies, using one easy to use, but powerful automation framework. The key and the purpose of this paper, is to help you understand how to fit all the tools together, to form your ready-for-anything toolbox.

The toolbox does not replace your existing tools; instead it helps you to utilize them more effectively. All the information, links and implementation examples are provided, so that you can have the flexibility to reach into your toolbox for whatever mix of technology comes your way to test.

Biography

James Gibbard is a Production Operations, QA Technical Lead at Intel Security, currently working at the Intel Security site in Aylesbury, Buckinghamshire, UK. Over the last 10 years, he has been involved in software testing of security software and web analytic solutions. This has ranged from hands-on testing of anti-malware engines with live virus samples, to managing QA teams, to setting software test vision and strategy for entire departments.

James has a BSc (Hons) in Internet Computing from the University of Hull, UK and is an ISTQB Advanced Level Test Manager.

1 Introduction

Software development teams are being asked to deliver value at increasingly faster rates, and to higher quality than ever. At the same time, software developers are making use of much more complex software stacks in order to supply this value to the customer, on time and to budget.

Software stacks comprise of layers of programs that work together towards a common goal. These layers are separated by logical boundaries, usually that coincide with the groups main function or purpose (e.g. database/storage layer). Each layer can be said to work 'on top' of the other, producing a stack.

The problem this paper sets out to help solve is the software tester's ability to effectively test this tornado of software, requirements and time constraints, in a flexible and pragmatic way.

Many software companies make use of large, expensive COTS (Commercial Off The Shelf) test automation tools. These typically don't have the adaptability to cope with the increasing complexity of today's software development stacks.

This paper will provide an approach and tooling, that can be used instead of these COTS applications.

The approach and tools have been used successfully and have helped to provide a great foundation for automated testing, which has led to multiple projects delivering with high quality.

2 The Challenges

Software test departments that have purchased large test automation tools are starting to face problems, when asked to test a project that uses technology that is not normally used.

The test automation engineers are then normally faced with the daunting task of coaxing the tool to test aspects of the software that it doesn't support out of the box.

2.1 The 'Thin Vertical Slice' Challenge

The main challenge with COTS test tools, are that they normally specialize in one type of test automation or programming language (e.g. it can only test C# developed software applications). This is no good when a test department or agile team is asked to test a software application that uses multiple technologies (not just C#).

For example, many of today's web applications are made up of the following:

Each of the layers needs testing separately, and then the entire solution needs testing to ensure it all fits and works together as expected. This is a daunting proposition if your test automation suite only supports testing one or two of the technologies used in this stack.

Due to the cost of most COTS test tools, the test department or agile team, may be faced with writing custom code or extensions for the suite, as there may not be budget to start from scratch.

2.2 Cost of Maintenance

It's a technically challenging task to modify a COTS test automation suite, to test technologies it's not designed to. It's even more challenging to maintain that custom code over an extended period of time.

At any point the software under test changes, it often means that the custom code needs to be updated to ensure compatibility. This all makes for an expensive and labor-intensive solution to a problem that will continue to cost money and time.

2.3 Record & Playback Won't Cut It Any Longer

There is an ever increasing set of situations where the Record & Playback approach offered by UI test automation tools doesn't provide the level of inspection needed to properly validate a complex software application.

One such situation may be if you wanted to test a Web UI's ability to correctly enter data into a database. If there was no method of validating the inputs via the UI, a record and playback approach would only allow you to validate that the UI didn't return an error.

This just isn't good enough, especially for teams that are developing in an agile manner, who may not have all the functionality in each sprint, with which to validate against.

3 Test with a Toolbox Approach

The toolbox approach is just that, an approach. It's not a tool, it's not a framework, its not even something you can download. It could even be called a philosophy.

It's actually an approach to writing automated test cases that provides the ability to, for example, combine UI testing with DB testing. It has the ability to test various levels of the software stack in isolation, but also provides a flexible approach of the testing of the layers together.

3.1 Toolset Requirements

Although it was just said that it's not a framework, a toolset is needed to help implement the approach. The following is a list of requirements, which will help to ensure that the toolbox approach can be used successfully.

1. Be usable in multiple testing situations (Smoke tests, Functional tests, Regression tests, Acceptance tests, simple tests, and complex tests)
2. Be able to be used for testing multiple types of software and applications (e.g. Command-line tool testing, GUI testing, HTTP/REST testing, etc.).
3. Be able to aggregate the results of the tests into an easy to understand and presentable form automatically.
4. Be able to be plugged into a CI (Continuous Integration) system to execute test runs (or similar test runner) (e.g. Jenkins)
5. Be flexible enough to run on a test/dev box (standalone), and on a complex remote test automation system.
6. Be able to allow test case reviews by people who may not understand programming. This can be helped by having clear, concise and human readable test cases.

3.2 Introducing Robot Framework

In the example used for the basis of this paper, we used Robot Framework as the main toolset that enables the toolbox approach.

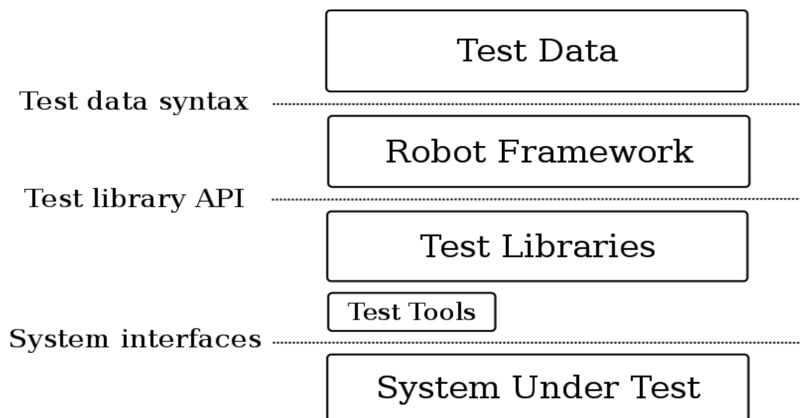
“Robot Framework is a generic test automation framework for acceptance testing and acceptance test-driven development (ATDD). It has easy-to-use tabular test data syntax and it utilizes the keyword-driven testing approach. Its testing capabilities can be extended by test libraries implemented either with Python or Java, and users can create new higher-level keywords from existing ones using the same syntax that is used for creating test cases.” (*Robot Framework*)

3.2.1 Multi-Platform Support

Robot Framework is compatible on both Linux and Windows, as it’s based on Python. It has a vast array of commands that work on both platforms. This is a great bonus for writing test cases once, which can work in any environment.

3.2.2 Modular Structure

Robot Framework’s power and beauty, and what makes the ‘toolbox’ approach possible, is its ability to allow users to create their own plugins or libraries that fit into the framework.



These libraries fit directly into Robot Framework by being loaded at the runtime of the test suite.

Libraries are called by your test suite by simply specifying the class name of your library as a configuration (*Robot Framework User Guide: Creating Test Libraries*).

3.2.3 Built-in Libraries

Robot Framework has many built-in libraries that can help to create test cases with no programming necessary. These include:

1. OperatingSystem (copy files, run commands, get environmental variables), great for running simple or complex system commands, that you might have done using a batch/bash script. (*Built-In Library Documentation: OperatingSystem*)
2. Collection (create list, add to list, remove from list), allows you to create arrays/lists of items that you test cases can make use of. (*Built-In Library Documentation: Collections*)
3. String (convert to lower case, get substring, get matching regex), provides powerful string manipulation functions without having to use awk or grep tools. (*Built-In Library Documentation: String*)

3.2.4 Custom Libraries

Custom libraries allow you to create your own keywords. These libraries can be implemented in Python, Java (using Jython) or C (using the Python C API).

3.2.5 Remote Libraries

Remote libraries are similar to the custom libraries, but they allow remote connections to external tools or applications that run in separate processes.

As an example (and one that I have used in the past), it is possible to create a remote library that can interact with a C# UI, by use of a remote C# service, that in turn uses the MS Automation Interface.

3.2.6 Combining Libraries

We have used both the Built-In and Custom libraries. They have proven very effective in allowing the creation of keywords that help to test the very complex stack of software used on our projects.

3.2.7 Other Benefits of Robot Framework

Some of the other numerous features and benefits of Robot Framework are:

Continuous Integration – Robot Framework integrates easily with Jenkins, using a freely available plugin. This provides result rollup, integrated logs and report, and much more.

Configuration Management – All test suites and test cases are saved as text files, custom libraries can be saved as Python files, so all elements can easily be version controlled.

Simple but powerful test result reporting/logging – One of Robot Frameworks best features, is its built in reporting. It rolls up all passed test cases, and automatically expands failed tests, to help aid investigation. An example is show in the following image.

```
[-] TEST SUITE: Invalid Login                                                        Expand All
    Full Name:             Login Tests.Invalid Login
    Documentation:         A test suite containing tests related to invalid login. These tests are data-driven by they nature. They use a single keyword, specified with Test Template setting, that is
                           called with different arguments to cover different scenarios.
    Source:                /private/tmp/Seleniumlibrary-demo/login_tests/invalid_login.txt
    Start / End / Elapsed: 20110721 13:13:16.517 / 20110721 13:13:22.254 / 00:00:05.737
    Status:                6 critical test, 5 passed, 1 failed
                           6 test total, 5 passed, 1 failed
    [+] SETUP: html_resource.Open Browser To Login Page
    [+] TEARDOWN: SeleniumLibrary.Close Browser
 [+] TEST CASE: Invalid Username                                                     Expand All
 [+] TEST CASE: Invalid Password                                                     Expand All
 [+] TEST CASE: Invalid Username And Password                                        Expand All
 [+] TEST CASE: Empty Username                                                       Expand All
 [-] TEST CASE: Empty Password                                                       Expand All
    Full Name:             Login Tests.Invalid Login.Empty Password
    Start / End / Elapsed: 20110721 13:13:21.683 / 20110721 13:13:21.895 / 00:00:00.212
    Status:                FAIL (critical)
    Message:               Location should have been 'http://localhost:7272/html/error.html' but was 'http://localhost:7272/html/welcome.html'
    [+] SETUP: html_resource.Go To Login Page
    [-] KEYWORD: Login With Invalid Credentials Should Fail ${VALID USER}, ${EMPTY}
      Start / End / Elapsed: 20110721 13:13:21.757 / 20110721 13:13:21.895 / 00:00:00.138
      [+] KEYWORD: html_resource.Input Username ${username}
      [+] KEYWORD: html_resource.Input Password ${password}
      [+] KEYWORD: html_resource.Submit Credentials
      [-] KEYWORD: html_resource.Login Should Have Failed
        Start / End / Elapsed: 20110721 13:13:21.880 / 20110721 13:13:21.894 / 00:00:00.014
        [-] KEYWORD: SeleniumLibrary.Location Should Be ${ERROR URL}
          Documentation:         Verifies that current URL is exactly `url`.
          Start / End / Elapsed: 20110721 13:13:21.881 / 20110721 13:13:21.894 / 00:00:00.013
          13:13:21.894  FAIL  Location should have been 'http://localhost:7272/html/error.html' but was 'http://localhost:7272/html/welcome.html'
```

3.3 High-level, Abstracted Language Automation Frameworks

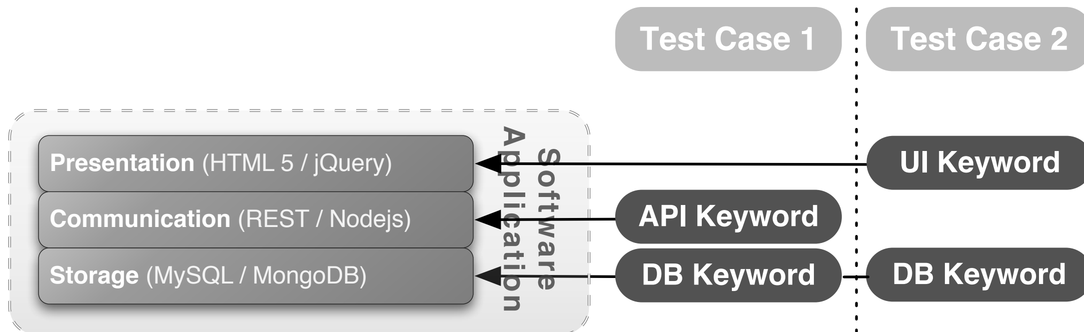
The whole reason why the toolbox testing approach is possible, is due to the introduction of high-level, abstracted language automation frameworks, such as Robot Framework and Cucumber (*Cucumber*).

They provide the ability to use keywords derived from different libraries or tools, in the same test cases. These test cases can then start to interact with the various levels of a software application (or stack).

What's more, due to the abstracted language that the test cases are written in, it allows the whole team (even members who are not familiar with code) to review, learn and suggest changes to automated test cases. This opens up a whole new perspective on test case reviews, and provides confidence that the test cases are testing the correct things, in the correct ways.

Additionally, in Robot Framework, test cases or parts of test cases can be re-used in multiple places. This provides a solution to keeping maintenance overheads down of commonly used test steps.

For example, you may have a test case that was initially created for testing an API & database, which can be re-used as part of a functional test case, once a UI is created. In the following diagram the 'DB Keyword' is re-used in Test Case 2.



4 Solving the 'Thin Vertical Slice' problem

As described earlier, the increasing uptake of agile in software development and the pressure on teams to provide value is getting greater. Most teams try to provide value by focusing on a thin vertical slice of the final software product.

Now that we have our framework, our toolbox can start to be developed, to help us test each vertical slice created by the team.

4.1.1 Identify & Standardize on Testing Tools

Firstly, start to identify your own team's software stack, and decide which tools you'd like to use to test them. This will require some level of standardization, but this is a good activity and can help to ensure that testing is done to a consistent level across your team.

Tools can range from command-line tools (e.g. network latency emulators) to Python libraries that are not part of your test automation framework already. You may also feel that the libraries/tools available built in to your framework (e.g. Robot Framework) or available online, are adequate enough for your testing (e.g. Selenium Library).

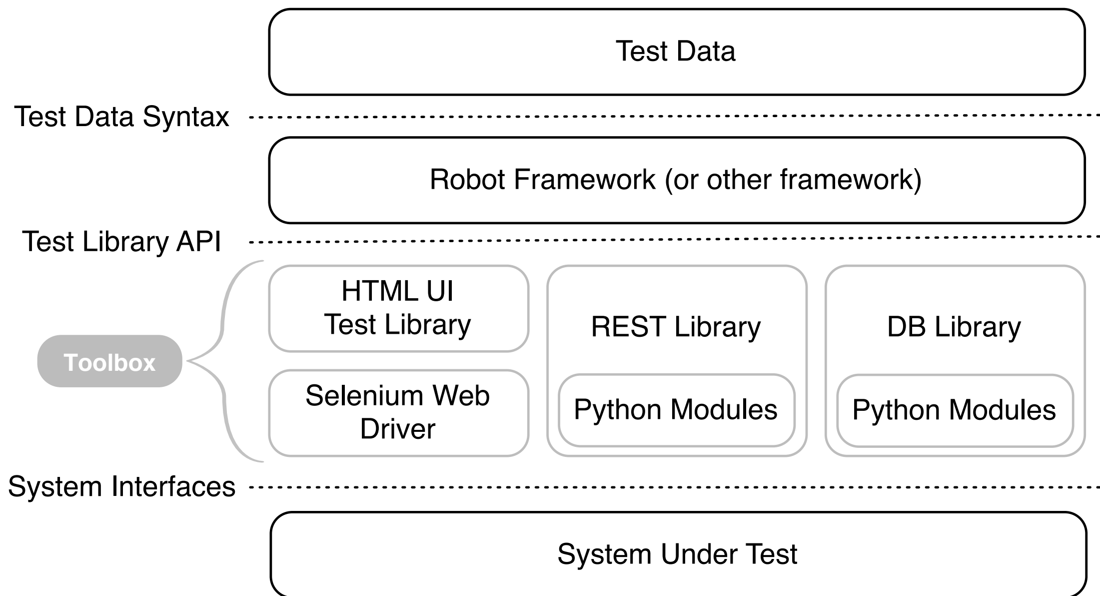
Copies may not be made or distributed for commercial use

4.1.2 Build Your Tools / Libraries

If you are going to write your own tools/libraries, once you've worked out which tools you are going to use, start to identify the ways that you want to interact with them. Which keywords are you going to need to utilize your test tool? You could use examples from other libraries and apply them to your tool.

In Robot Framework, building libraries involves creating Custom Libraries which can be imported into your test suite. (*Robot Framework User Guide: Creating Test Libraries*)

The diagram below shows how your toolbox contains various libraries, that are utilizing different technologies and tools. These are all loaded into Robot Framework and can now be called from your test cases.



4.1.3 Putting it all together

The key to the toolbox approach is to start to utilize your tools/libraries together when creating your test cases. Import your library for interacting with the API and import your library for connecting to the database.

Now you have the ability to easily validate that data inputted via the API is correctly stored in the database without having validate output on the UI (which may not have been implemented yet).

The following is an example of a Robot Framework test suite, containing two test cases.

```
*** Settings ***
Library     OperatingSystem
Library     Database_Lib
Library     UI_Lib
Library     Common_Lib
```

Global settings, including importing your built-in and custom libraries

```
*** Variables ***
${API-URL}  http://localhost/api/user/create
${UI-URL}http://localhost/ui/users
```

Global variables that can be reused in multiple test cases

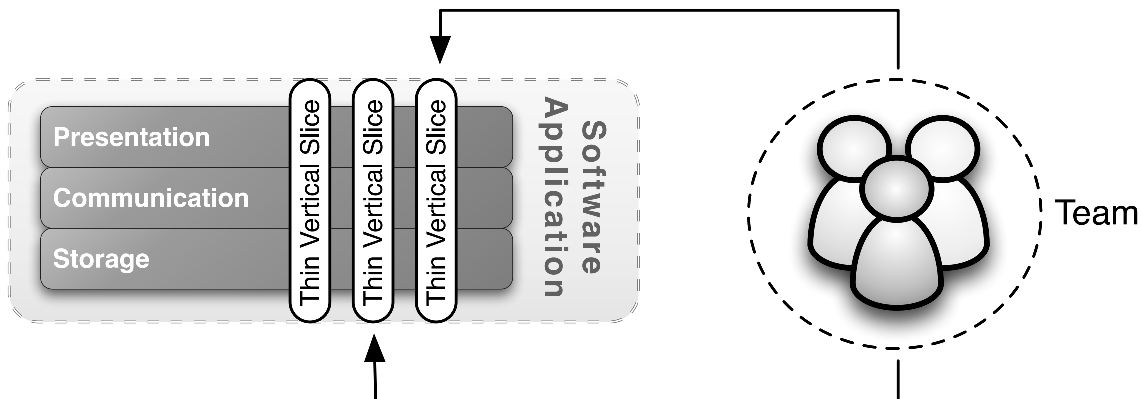
```
*** Test Cases ***
TC01 - Verify new user is created (in Database) using API
   [Tags] API Regression
   Create User    qa-user     ${API-URL}
   Verify User in Database    qa-user
   [Teardown] Purge User from Database    qa-user

TC02 - Verify new user is created (in UI) using API
   [Tags] API Regression
   Create User    qa-user     ${API-URL}
   Open Browser   ${UI-URL}   Chrome
   Verify User in UI    qa-user
   [Teardown] Purge User from Database    qa-user
```

Test Case 01: Initially created to test the API & DB

Test Case 02: Reusing the 'Create User' keyword from TC01, but testing the UI

The whole team now has the agility to start testing elements (slices of the final solution) in an efficient and effective way.



4.1.4 Fill in the gaps / Augmenting your test cases

In some cases, a library may not always be as reliable or as able to interact with a software application as you would like (this could be due to bugs or poorly implemented functionality). In this case its possible to make use of another library to fill in the gap (i.e. augment your test case). This can ensure that you are efficient at getting test cases written and software tested on time. You can then return to the test case at a later stage to make it more robust, once bugs have been fixed in your primary library or when better features exist.

For example, your C# UI Test library is unable to click and drag UI elements to certain coordinates on the application under test. Having another library, using a very different technology (e.g. Sikuli, an image recognition and UI interaction technology developed by MIT) (*MIT. 2015*), you are able to import the Sikuli library (as well as your C#), which you can use for the click and drag step. Then switch back to your C# library for the remainder of the test case.

5 Future Direction

Due to the simplicity and readability of the libraries that can be created in these high-level, abstracted language test frameworks, future usages could include:

- Allowing customers or business stakeholders to be involved in test case reviews (of automated tests).
- Using the framework and custom UAT (User Acceptance Test) libraries, automated UAT tests could be created by the user.

Copies may not be made or distributed for commercial use

- Due to using one tool to test various technologies, a team will be able to write test cases faster instead of having to learn different test tools for different layers of the stack.
- Commonly used libraries and keywords can be refactored to be globally usable, providing a vast array of keywords to use on new projects.

6 Conclusion

In conclusion the Toolbox testing approach can enable teams to have a flexible, pragmatic approach to writing automated test cases, while still being able to adapt to changing requirements.

It no longer has to be a daunting task to create test automation for complex stacks of software.

With the ability of Robot Framework (and other keyword driven frameworks) to write automated tests in an easy to understand and reviewable format, it starts to break down the barriers to test case automation.

This approach has really helped teams I've worked with to implement testing that provides value and at the same time achieve high levels of quality.

Copies may not be made or distributed for commercial use

References

Robot Framework, http://robotframework.org/ (accessed June 1, 2015).

Robot Framework User Guide: Creating Test Libraries,
http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html - creating-test-libraries
(accessed June 1, 2015)

Robot Framework User Guide: Using Test Libraries,
http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-test-libraries
(accessed June 2, 2015)

Built-In Library Documentation: OperatingSystem,
http://robotframework.org/robotframework/latest/libraries/OperatingSystem.html (accessed June 2, 2015)

Built-In Library Documentation: Collections,
http://robotframework.org/robotframework/latest/libraries/Collections.html (access June 2, 2015)

Built-In Library Documentation: String,
http://robotframework.org/robotframework/latest/libraries/String.html (access June 2, 2015)

MIT. 2015. “Sikuli Script”, http://www.sikuli.org (access June 5, 2015)

Cucumber, https://cucumber.io/ (accessed June 7, 2015)