University of Victoria
Engineering & Computer Science Co-op
Work Term Report
Spring 2020

Creating Unit Tests for Cloudscheduler's Web Interface

University of Victoria
High Energy Research Computing
Victoria, B.C., Canada

Garet Robertson
V00882812
Work Term 1
Computer Science
grobertson@uvic.ca
April 27, 2020

In partial fulfillment of the academic requirements of this co-op term

Signature: R Seusl Position: HEPNET Technical M. Date: 27/04/20

Name (print): Rolf Seuster E-Mail: seuster@uvic.ca

For (Company Name)____________________

Abstract

When I started working for the High Energy Physics Research Computing (HEPRC) at the University of Victoria, Cloudscheduler, a software system that they developed and maintained, had unit tests for two of its three application programming interfaces (APIs). I was given the task of creating a test framework and some unit tests for the third interface. I considered several frameworks to automate web browser interaction, and several others to facilitate the execution of unit tests. After settling on Selenium and Unittest for these roles, I created common functions to automate the assertion of the presence of elements and submission of forms. I created thorough unit tests for a few of the web interface pages, but was unable to do so for the rest because of the short-term nature of my position with HEPRC.

Report Specification

Audience

The intended audience is others who wish to run or improve the unit tests of Cloudscheduler, whether or not they are working with High Energy Physics Research Computing as I was.

Prerequisites

An understanding of basic computing terms (such as server, network, etc.) is required.

Purpose

This report documents the reason unit tests were created for Cloudscheduler’s web interface as well as the methodology used to create them. It may be useful to anyone who wishes to learn about, run, or improve these tests.

Table of Contents

Glossary

application programming interface (API)	A standardized interface for a program that can be used to connect it to other programs.
command line interface (CLI)	A user interface that consists only of text. The user gives commands which are executed and may produce output.
graphical user interface (GUI)	A user interface that uses graphics to communicate with the user.
Hypertext Transfer Protocol (HTTP)	A protocol that defines a way to send information over the internet.
JavaScript Object Notation (JSON)	A human-readable data interchange format.
job	A computing task that can be outsourced.
Python	A high-level computer programming language that supports both functional and object-oriented programming.
test case	A single test defined by required test conditions, a set of inputs, and a set of expected outputs.
test suite	A collection of related test suites and / or test cases which may also include setup or tear down procedures.
virtual machine (VM)	A software simulation of a physical machine that can operate on a different machine which may have a different architecture and / or operating system.
unit test	A small program or section of a program that is created to test a small component or function of a software system in isolation from the rest of the system.

Introduction

Cloudscheduler

Cloudscheduler is a Python program that manages the booting and shutting down of virtual machines (VMs) on servers, which it refers to as clouds, as necessary to facilitate the efficient execution of jobs in these VMs [1]. A server that is running Cloudscheduler and a few other programs (which must be configured to communicate with Cloudscheduler) is known as a Cloudscheduler server. A client submits jobs, along with information about the resources that should be allocated to run them, to a Cloudscheduler server, which initially stores these jobs and metadata in a queue in a database. Cloudscheduler queries the database and searches for clouds it is connected to that have the hardware and software necessary to run the jobs in the queue. If these clouds already have VMs running that match the specifications, Cloudscheduler will run the jobs in these VMs. Otherwise it will boot new VMs as necessary. If a VM is idle for a certain (configurable) length of time, Cloudscheduler will shut it down to conserve resources.

To keep jobs organized and allow different levels of control to different people, Cloudscheduler defines groups and users, the details of which are also stored in the database. All actions are performed by users. Users may be in any number of groups, but must be in at least one to perform any actions. Only super users can create and delete groups and users. Each cloud belongs to a single group, and clouds may be created, modified, and deleted by any of the group’s users. A client may query and control Cloudscheduler servers over the internet in three different ways: by sending Hypertext Transfer Protocol (HTTP) requests which contain and request JavaScript Object Notation (JSON) data, by using the Cloudscheduler command line interface (CLI) (if it is installed on the client machine), or by using the web interface. Sending HTTP requests invovles managing a few tokens and credentials that must be sent in each request. Therefore, it is meant to be a low-level application programming interface (API) which may be built upon. The CLI does just this, keeping track of the tokens and credentials, and providing a more user-friendly interface. It is also includes documentation for nearly every command one may use. The web interface, which is viewed in a web browser, operates on and displays the same data that the CLI delivers, but through web pages. The main status page refreshes regularly, giving one a live overview of the many VMs running on many clouds simultaneously. A view of the status page is given in the appendix.

Cloudscheduler is maintained and improved by High Energy Physics Research Computing at the University of Victoria. In my work experience position there I improved unit tests which already existed for the HTTP API and CLI, and designed and created unit tests for the web interface. In this report I will explain how the existing test framework worked, the options I explored when designing the framework for the web interface unit tests, and how I implemented the web interface tests.

Existing Tests

Unit tests already existed for the HTTP API and the CLI. These were created and run using a custom framework. The main program, which was written in Python, ran all Python programs in its present directory that matched a particular naming scheme. These were assumed to be test suites. In each HTTP test suite, a single function was called many times, each time submitting a request and expecting a response matching given criteria. A similar function existed for CLI commands. Both would print the details of the test and its result (whether success or failure). Examples of HTTP and CLI test cases are given in the appendix. This system was built from scratch, not using any external test framework. This was justifiable, because the tests only varied in their inputs and expected outputs, and a custom test framework permitted the abstraction of nearly the entire test execution process. This technique did, however, restrict the types of expectations that could me made about a response when writing test cases. For example, in HTTP tests it was not feasible to assert that an object did not appear in a list of objects returned by the server, nor to assert how many objects appeared in such a list.

Options Explored

Testing Frameworks

Because both Cloudscheduler and the existing unit tests were written essentially entirely in Python, it made sense to only explore Python testing frameworks.

Existing Test Framework

Using the existing framework would not have worked well, because it was designed to use a single function that would take a collection of inputs and and expected outputs, perform a static list of actions, and report on whether the expected outputs matched the actual. In the case of the web interface, the actions that need to be performed are different for each page and there are multiple ways to give inputs (through clicking and typing). Certain actions must be separated by a waiting period so that the web browser waits for a new page or dynamically generated elements to load before making assertions about their contents. For these reasons it would be difficult and cumbersome to encode all types of web actions and all elements expected to be present on each page in Python objects.

Unittest

Unittest is a testing framework that is part of the Python standard library [2]. As a result it is widely used and likely to be supported far into the future. It is an xUnit framework, making it easier for someone who has used another xUnit framework such as JUnit to understand and use it, even if they do not have much experience with Python. A side effect of this is that Unittest uses many assert methods (such as assertEqual, assertFalse, assertIsNone, assertNotIn, and assertRaises). These methods must be remembered or looked up in a reference if Unittest is to be used according to its design. It groups similar tests into classes, and allows execution of individual tests from the command line. Considering these pros and cons, I chose to use Unittest as the framework for the web interface tests.

Pytest

Another popular and well-regarded Python test framework is Pytest [3]. It is not part of the standard library, meaning that anyone who wished to run the tests would have to install it. Pytest tests are often more compact than unittest tests, though neither suffer from verbosity. It supports test parameterization using decorators, meaning a decorator can be placed above a function to specify that it should be run with multiple inputs, as well as specifying the expected output for each. While some of the web interface tests did require performing the same operation for several inputs, each of these inputs was usually a fairly large dictionary, meaning it would be awkward to put it all into a decorator. The inputs were also assembled as combinations of parameters from a different dictionary. Pytest also uses the built-in *assert* keyword for all assert statements rather than using many assert methods as Unittest does. Although Pytest has advantages that Unittest lacks, I believed Unittest was the better tool for the job.

Browser Automation Tools

Selenium

Selenium is a collection of frameworks that implement the WebDriver specification for automating web browser interaction in several languages, one of which is Python [4]. Selenium is among the most popular web automation frameworks for Python, and is one of the only ones that creates a full instance of a third-party web browser with which to interact. In addition to giving it support for JavaScript, this allows it to catch problems that are caused by the graphical user interface (GUI), such as an element being unclickable because it is covered up by another element. For these reasons it was chosen to automate browser interaction.

Mechanize

Another web automation option for Python was Mechanize [5]. Mechanize automates the filling of forms and has a system to find and click on links. However, it implements its own low-feature browser rather than connecting to a well-known one. As a result it does not it does not support JavaScript, which is used in the web interface, making it impratical to use in the tests [6]. It would also have prevented me from testing how browsers that I expect the end user to actually use display the interface. Forms and links are the only elements that it provides functionality for, so testing other elements such as the content of a table would require another tool or manual HTML parsing.

Zope.testbrowser

Zope.testbrowser was created for Zope, web application server software written in Python [7]. It does support form submission, but similarly to Mechanize it does not parse other HTML elements or create objects that can be interacted with as Selenium does. This would force me to use another tool to parse other elements. Zope.testbrowser also lacks a GUI, meaning it is unable to find GUI-related issues as Selenium is.

Solution Implemented

Setup

The HTTP tests had a different setup routine for each object. When executed through a connection to one of HEPRC's Cloudscheduler development servers, each setup routine took usually between 20 and 60 seconds to run. The CLI tests had only one setup routine, but since this had to create more test objects it took closer to 80 seconds. These times are usually independent of the client machine, because it spends the vast majority of its time waiting for the server. Since the web interface setup had to start a web browser and wait for it to load a webpage in addition to destroying old test objects and creating new ones, it was expected to take longer. It was therefore desirable to run it once at the beginning of testing rather than for each web interface page. I added a boolean value to the existing credentials file to indicate whether the tests were currently setup or not. This boolean was checked before tests for each page were run, and setup was run if necessary. I also added a utility script to run the clean up (when one was finished testing) or flip the boolean in the credentials file (which would trigger a clean up and setup, which is necessary when running certain tests twice in a row).

Firefox Profiles

Cloudscheduler uses HTTP basic access authentication to restrict access to the web interface to Cloudscheduler users. This causes the browser to open a particular kind of dialog box with username and password fields. The ability to input arbitrary text into these fields is not included in the the WebDriver specification, and therefore is not supported by Selenium [8]. This was solved by requiring one who wishes to run the tests to create a Firefox profile for each test user that performs actions that has the credentials of this user saved in it. They are then prompted for the paths of the directories where these profiles are saved the first time they run the web interface tests (at which time the test credentials file is automatically created). When preparing to run tests, these paths are provided to WebDriver to create the instances of Firefox in which the web interface is tested. Firefox automatically enters the saved credentials into the prompt, and WebDriver accepts the prompt (which is equivalent to clicking "OK").

I investigated whether these profiles could be created automatically (since the test framework has all of the credentials), but Firefox encrypts the credentials. I was unable to determine the encryption method Firefox uses, and based on the complexity of tools I found to decrypt these passwords, I decided that the cost of implementing the encryption would likely not be worth the simplicity it would allow in running the tests. Three test users attempt to perform actions and therefore require profiles: a non-super user, a super user, and a user who is not in any groups.

Conclusion

I constructed the framework to test the web interface, including common functions to assert the existence of elements and submitting forms, and implemented tests for two of the pages. However, because of the limited (four-month) length of my position with HEPRC, I was unable to implement tests for the other eight pages. I do not forsee a significant number of other common functions needing to be written, so I believe the task of implementing the other tests would be straightforward.

Acknowledgements

I would like to thank Colin Leavett-Brown, Colson Driemel, and Rolf Seuster for their patience in helping me debug problems and teaching me, and for giving me the freedom to make decisions about the implementation of the web interface tests.

References

1. C. Leavett-Brown, R. Seuster, M. Ebert, M. Paterson. “Cloudscheduler/README.md at dev · hep-gc/cloudscheduler.” https://github.com/hep-gc/cloudscheduler/blob/dev/README.md (accessed Apr. 20, 2020).

2. E. Melotti, et al. “Unittest — Unit testing framework — Python 3.8.2 documentation.” Python. https://docs.python.org/3/library/unittest.html (accessed Apr. 15, 2020).

3. R. Pfannschmidt, et al. “Pytest: helps you write better programs — pytest documentation.” Pytest. https://docs.pytest.org/en/latest/ (accessed Apr. 15, 2020).

4. D. Burns, et al. “Selenium Documentation — Selenium 3.14 documentation.” Selenium. https://www.selenium.dev/selenium/docs/api/py/api.html (accessed Apr. 20, 2020).

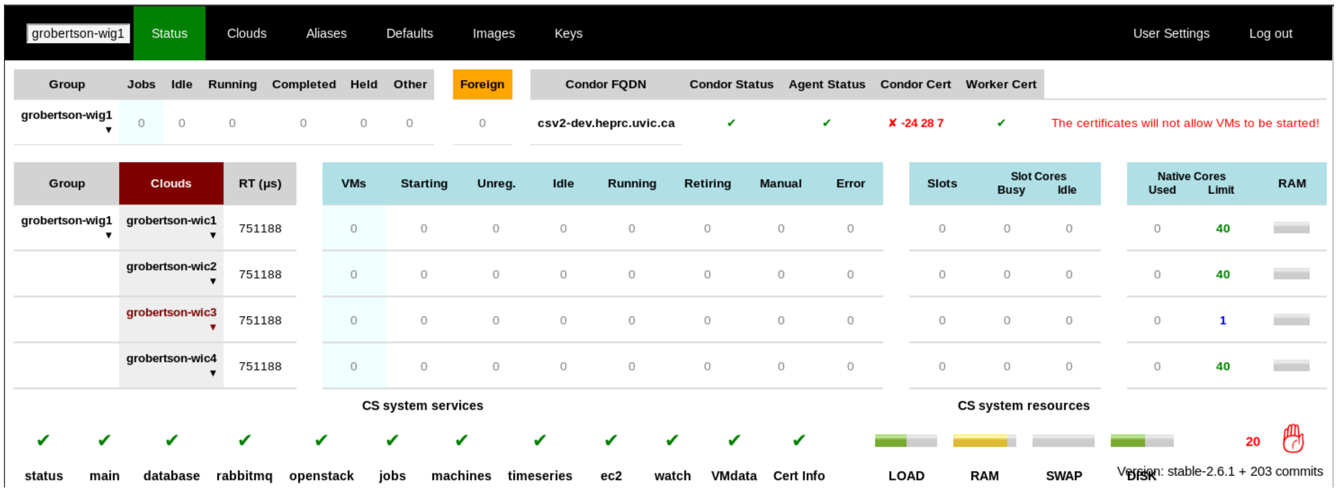
5. K. Goyal, et al. “Mechanize — mechanize 0.4.5 documentation.” Read the Docs. https://mechanize.readthedocs.io/en/latest/ (accessed Apr. 15, 2020).

6. K. Goyal, et al. “Frequently Asked Questions — mechanize 0.4.5 documentation.” Read the Docs. https://mechanize.readthedocs.io/en/latest/faq.html#jsfaq (accessed Apr. 15, 2020).

7. C. Watson, et al. “Zope.testbrowser — zope.testbrowser 5.0 documentation.” Read the Docs. https://zopetestbrowser.readthedocs.io/en/latest/index.html (accessed Apr. 15, 2020).

8. A. Tolfsen, et al. “Missing support for HTTP authentication prompts · Issue #385 · w3c/webdriver.” GitHub. https://github.com/w3c/webdriver/issues/385 (accessed Apr. 20, 2020).

Appendix



grobertson-wig1 | Status | Clouds | Aliases | Defaults | Images | Keys | User Settings | Log out

Group	Jobs	Idle	Running	Completed	Held	Other	Foreign	Condor FQDN	Condor Status	Agent Status	Condor Cert	Worker Cert
grobertson-wig1 ▾	0	0	0	0	0	0	0	csv2-dev.heprc.uvic.ca	✓	✓	✘ -24 28 7	✓

The certificates will not allow VMs to be started!

Group	Clouds	RT (μs)	VMs	Starting	Unreg.	Idle	Running	Retiring	Manual	Error	Slots	Slot Cores Busy	Slot Cores Idle	Native Cores Used	Native Cores Limit	RAM
grobertson-wig1 ▾	grobertson-wic1 ▾	751188	0	0	0	0	0	0	0	0	0	0	0	0	40	
	grobertson-wic2 ▾	751188	0	0	0	0	0	0	0	0	0	0	0	0	40	
	grobertson-wic3 ▾	751188	0	0	0	0	0	0	0	0	0	0	0	0	1	
	grobertson-wic4 ▾	751188	0	0	0	0	0	0	0	0	0	0	0	0	40	

CS system services

status	main	database	rabbitmq	openstack	jobs	machines	timeseries	ec2	watch	VMdata	Cert Info
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

CS system resources

LOAD RAM SWAP DISK 20

Version: stable-2.6.1 + 203 commits

Figure 1: The status page of the web interface.

```
# 07
execute_csv2_request(
    gvar, 1, 'GV', 'request contained a bad parameter "invalid-unit-test".',
    '/group/list/', group=ut_id(gvar, 'gtg4'), form_data={'invalid-unit-test': 'invalid-unit-test'},
    server_user=ut_id(gvar, 'gtu5')
)
```

Figure 2: An example of an HTTP test case.

```
# 16
execute_csv2_command(
    gvar, 1, None, 'The following command line arguments were invalid: cloud-name',
    ['group', 'list', '-cn', 'invalid-unit-test', '-g', ut_id(gvar, 'clg1')]
)
```

Figure 3: An example of a CLI test case.