UROP Web Application Testing Report

Ruijing Zhang
University of Melbourne

Introduction

This UROP project explores web application test automation. The subject of the testing is GenomeSpace, a cloud-based storage site acting as a bridge between the Nectar Research Cloud and Genomics Virtual Laboratories. GenomeSpace allows users to mount Swift containers from their Nectar cloud, and to send the files in their containers to the virtual laboratories for processing. GenomeSpace also provides basic file manipulation functionality such as moving, copying, renaming, deletion.

Background

Faulty application functionality can lead to customer losses and even potential financial loss, therefore making sure that the application works correctly is a crucial part in the development of any software application. As a web application, GenomeSpace has a very high dependency on the network: any change in the network applications to which GenomeSpace is connected could cause failures in GenomeSpace itself. Manual testing of an application requires significant amounts of developer time, to ensure every piece of functionality is tested and to identify any faulty behaviour of the application, but there is a promising alternative: test automation, which can potentially relieve developers of the more tedious aspects of testing and allows them to focus more on the product itself.

Aim

The aim of this project is to complete an automatic testing program for GenomeSpace, to ensure that all its functionality works as intended, and that the bugs and regressions in functionality can be spotted easily, immediately and with minimum cost of resources. The ultimate goal is to put the program to use for aiding the development and maintenance of the application.

Methodology

The testing program is written in Python, and makes use of libraries: Selenium and Unittest. Selenium provides functionality for simulating the website browsing process, such as the user opening a browser, going to a page, typing something in a text field and clicking a link to go to the linked page. It also provides functionality for such thing as halting the program until an alert is received and waiting for the page to be loaded. Unittest allows grouping the tests into test cases, one for each piece of functionality tested. These test cases are run consecutively, with their results shown on screen and a report based on the test results automatically generated after each run of the program.

The functionality tested at this stage includes:

- user registration
- user login
- mounting a swift container
- disconnecting a swift container
- importing a file using the expired public URL
- generating the public URL of a file and accessing the file with the URL
- file renaming
- copying a file between directories

- copying a file between containers
- moving a file between directories
- moving a file between containers
- deleting a file
- launching a GVL instance with files
- generating the DOI of a file for publishing

Results and Analysis

Initially, the user interface was tested with Selenium by conducting a series of event tests, waiting for the responses and checking whether the outcomes were as expected. In order to do this, the web page elements to which the events apply had to be located. When simulating a user login process, for example, the location of the username and password fields within the webpage had to be identified (shown in Figure 1). These particular elements could easily be located because each has a unique id associated with it.

```
                <label class="uname">USERNAME: <input type="text"
class="gslogininput" name="user_name" id="identity" value=""/></label>
                <br/>
                <label class="pword">PASSWORD: <input type="password"
class="gslogininput" name="password" id="password"/></label>
```

Figure 1. HTML Elements for Login Fields in GenomeSpace

Unfortunately however, the majority of elements within the page are dynamically generated and injected by Ajax. Due to how this is done such elements often don't have an id, name or anything else with which they can be uniquely identified. Some of them even consist solely of tag names. While this issues could be avoided if the backend code was developed alongside the testing infrastructure, in this case that was not possible as the UI-generation code had already been developed.
The buttons in the dialogue for renaming a file or a directory are shown in Figure 2.

```
▼<div class="dialogButtonDiv">
    <button>Rename</button>
    <button>Close</button>
  </div>
```

Figure 2. HTML Elements for 'Rename' and 'Close' Buttons in GenomeSpace

Query:	Results (1):
/html/body/div[@class='ui-dialog ui-widget ui-widget-content ui-corner-all ui-draggable ui-resizable'][20]/div[@class='ui-dialog-content ui-widget-content']/div[@class='dialogButtonDiv']/button[1]	Rename

Figure 3. Xpath of the 'Rename' Button in Figure 2.

```
[20]/button[1]
```

Figure 4. Two Indexing Numbers in the Xpath in Figure 3.

It is impossible to locate the elements without hardcoding some indexes, and hardcoded program are fragile and can break easily if the page layout is changed. This made testing of the whole user interface infeasible, hence the focus of the project move towards service testing instead.

The functionality of GenomeSpace is service oriented, consisting primarily of client-servers communication. In order to address the issue described above, another approach is adopted for test cases that, unlike registration and log-in, involve elements without unique identifiers: sending requests to the services and checking whether they return the correct responses.

The method used for response checking is analysis of each response's status code, using injected Javascript to generate alerts that Selenium can then act on. After an alert occurs, Selenium records and analyses its content then closes the alert, allowing test execution to continue.

In terms of waiting, there are many situations in which the user (or a program) needs to wait for the Internet or the browser to process a request. An example particularly pertinent to this project is page refreshing. The method currently used to handle this is sleeping for an arbitrary amount of time based on measurements made of the average and maximum refresh times, being careful to ensure that the wait is not too long so that minimal time is wasted. This approach is however not foolproof: such waiting times are quite dependent on network performance and Internet speed, which can sometimes be difficult to predict.

An alternative approach is to check every one second or two whether the awaited event is complete, recording the amount of time has been spent waiting. For events like refreshing a page or renaming a file, if the preset timeout is reached then test can be considered to have failed, as a quality application is not supposed to make the users wait too long for such simple events. With process like uploading or downloading large files, however, it is difficult to determine whether a request is still being processed or has timed out, as such requests can sometimes take even day or longer to complete. A potential solution would be adding an event listener to each of the upload and download functions, which listen to the "progress" of the request and signal a timeout of the transfer fails or ceases progressing.

Overall, the test automation allowed many service failures to be identified and reported immediately, even when the testing program was still in its development stage. CORS issues, for instance, were causing random differences in file uploading and downloading behaviour for different users.

Individual service testing of the kind conducted is however not without limitations. While it can identify service failures, it cannot uncover faulty links or buttons in the user interface; the functionality of getting the DOI for a file, for instance, works perfectly on the back-end but behaves unpredictably when a user clicks the relevant UI button, and the service testing conducted cannot predict this.

Conclusion

Test-automation is an effective method for quickly identifying bugs and regressions in functionality with minimum cost in terms of developer time. It is however of limited suitability for testing dynamically generated web user interface elements, although this limitation could potentially be overcome if the UI generation code was written with a focus on testability.