This UROP project is a web application test automation project. GenomeSpace, which is a cloud-based storage acting like a bridge connecting the Nectar Research Cloud with the Genomics Virtual Laboratories, is the website that the program is testing on. A GenomeSpace user can mount the swift container from his/her Nectar cloud and then send the files in the container to the virtual laboratories for processing. GenomeSpace also provide basic file manipulation functionalities like move, copy, rename, delete, etc.

Any faulty functionality in an application could cause customer loss and potential financial loss, therefore making sure that the application is working correctly is a crucial part in the development of any software application. GenomeSpace is a web application and has a very high dependency on the network. Any change in any part of the applications related to GenomeSpace could cause failures in GenomeSpace. A manual testing of the application requires a developer to spend huge amount of time to test through every functionality of the application and keep an eye closely on any response of the application, which is a huge cost of human and time resources and this is where test automation becomes important.

The aim of this project is to complete and provide an automatic testing program for GenomeSpace to ensure that all the functionalities are working as intended and that the broken ones could be spotted easily, immediately and with minimum cost of resources. Ultimately, the program would be put to use and aid the development and maintenance of the application.

The program is written in Python and the two libraries used in this program are Selenium and Unittest. Selenium provides functionalities of simulating the website browsing process, for examples open up a browser, go to a page, type something in the text fields and click a link to go to the linked page, and also provides the functionalities like halt the program until an alert is present or wait for the page to be loaded. Unittest is for putting the test into test cases, one for each functionality. The test cases are run one after another and the result of each one is shown. A report based on the test results will be auto generated after each run of the program.

The functionalities that are tested at this stage include:

* user registration
* user login
* mount a swift container
* disconnect a swift container
* import a file using the expired public URL
* generate public URL of a file and access the file with the URL
* file rename
* copy file between directories
* copy file between containers
* move file between directories
* move file between containers
* delete a file
* lunch a GVL instance with files
* generate a DOI of a file for publishing

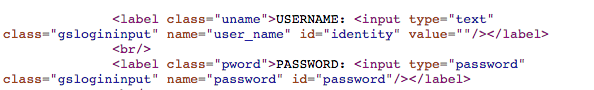
Initially, the user interface was tested with Selenium by conducting a series of events, waiting for the response and asserting the outcome was as expected. In order to do so, elements of the web page needed to be located for the events to be conducted on; for example when simulating a user login process, username and password fields are needed to be located in the webpage (shown in Figure 1). These locatings can be easily done because each of the elements has an id associated with. 

Figure 1. HTML Elements for Login Fields in GenomeSpace

However, most of the elements in the page are dynamically generated and injected by Ajax. They don’t have any id, any name or anything that can be uniquely identified. Some of them are even merely with tag names.

The buttons in the dialog for renaming a file or a directory are shown in Figure 1.



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

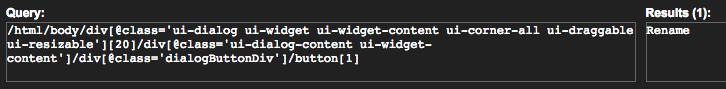


Figure 3. Xpath of the ‘Rename’ Button in Figure 2.



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

It is impossible to locate the elements without hardcoding some indexing numbers and the hardcoded program are always fragile and easy to break. That was why most of the test cases are not about the interface now.

As the functionalities of GenomeSpace are service oriented. The communications between clients and servers are the most crucial part. In order to solve the problem described above another approach is used for the test cases apart from registration and log-in, which is sending requests and checking if the response is correct.

The method of checking the responses is to get the response status codes in the alerts, giving by the injected JavaScript, using Selenium. Selenium library supports waits for different situations including waiting for the alert. After the alert shows up, switching to the alert and getting the alert content can also be done easily with Selenium.

In terms of waiting, there are many situations that a user (or a program) needs to wait for the Internet or the browser to process the request. An example particularly in this project would be page refreshing. The method using now is to sleep for an arbitrary amount of time and the time should be long enough to complete the process but cannot be too long that much time is wasted. Another problem comes; the waiting time has a very high dependency on the Internet performance and the Internet speed could be so random that is difficult to predict.

The alternative way of setting arbitrary mount of sleeping time is to make the program wait and check every one second or two meanwhile record the amount of time has been spent on waiting. For events like refreshing a page or renaming a file, if the preset timeout is reached then fail the case as a good application is not suppose to make the users wait for such long time anyway. However with process like uploading or downloading a file with a huge size, it is difficult to tell if the request is still being processed or not since those requests sometimes even require a day or more. A solution could be adding a event listener to each of the upload and download functions, which listen to the “progress” of the request and set a timeout in those listeners.

With test automation, many service failures were spotted and reported immediately even when the testing program is still at its development stage. For instance, the CORS problem caused random behaviours between different users for file uploading and downloading. The single service tests by sending HTTP requests can identify a service failure well, however, it cannot spot any faulty links or buttons in the user Interface; for instance, the functionality of getting DOI for publish a data file works perfectly on the back-end when tested but when a user click the button on the Interface, random number of requests are sent at once.

In conclusion, test-automation is an effective method for quickly identifying bugs and regressions in functionality with minimum cost in terms of developer time. However, the service testing has limitations on finding user interface problem and the integrity web application test would be best to start from the front-end user interface with the ids are provided for each important elements in the webpage.