

COMP 396

Project Zombie: Automated Testing of the Opal Application

James Mesich

April 2018



Project Zombie: Automated Testing of the Opal Application

James Mesich

*School of Computer Science
McGill University, Montreal, Canada
james.mesich@mail.mcgill.ca*

*McGill University Health Centre
Cedars Cancer Centre
1001 Decarie Blvd.
Montreal, QC
Canada, H4A 3J1*

COMP 396 Final Report

Approved for public release; distribution is unlimited.

Abstract: The OPAL oncology application is a mobile app developed by Health Informatics Group (HIG) for patients at the MUHC Glen hospital. This paper details the development of the automated testing suite for the Opal application.

Table of Contents

Figures and Tables	iii
Acknowledgements	iv
Abbreviations	iv
1 Introduction	1
2 Background and Related Work	1
3 Software Design	2
3.1 Preparatory Programs	2
3.1.1 <i>HTML Modifications</i>	2
3.2 Testing Suite	3
3.2.1 <i>Test Runner</i>	3
3.2.2 <i>Page Functions</i>	3
3.2.3 <i>Individual Page Tester</i>	4
3.3 Executing the test	5
4 Limitations and Future Work	5
4.1 Limitations	5
4.2 Future Work	6
References	7
Appendix A: Screenshots	8

Figures and Tables

Figures

Figure 1. HTML custom attributes	2
Figure 2. General List: Landing View	4
Figure 3. Select List	4
Figure 4. User Input: Login View	5
Figure A1. User Input: Login View	8
Figure A2. User Input: Login View	8
Figure A3. HMTL custom attributes	8
Figure A4. General List: Landing View	9
Figure A5. Select List	9

Tables

Acknowledgements

I would like to thank Laurie Hendren and the HIG team for the opportunity to be a part of the development of the Opal Application.

Abbreviations

MUHC: McGill University Hospital Centre

HIG: Health Informatics Group

DOM: Document Object Model

1 Introduction

The oncology portal and application (Opal) is an application designed for current oncology patients at MUHC developed by the HIG. The goal of this application is to provide the patients with protected health information and educational materials and help the patient become more active in their own treatment and thus improve their experience at the cancer center.

As the Opal application continues to add new modules and functionality, manually testing the structure and front end functionality becomes extremely time-consuming and inefficient. As a result, a testing suite needed to be developed. This testing suite works its way through the application in a similar way that a user would and tests whether the structure of the application is intact and basic functionality as well. The main goal of these tests are to ensure that the application works as expected from the user's point of view.

This report discusses the development of the testing suite designed for the Opal application. Section 2 looks at existing ideas, tools and libraries that were used in the development of the testing suite. In Section 3, it details the process of modifying the Opal application to make it ready to be tested all the way to the implementation of the testing suite. Finally, Section 4 discusses the limitations of the testing suite as well as the future work that can be done to improve the testing suite.

2 Background and Related Work

There are many web application testing and mobile testing tools available to use for many different use cases. However, the Opal application doesn't fit nicely into one or the other as it is a hybrid application both of mobile and web. While no one testing framework is meant for this, Selenium WebDriver was able to provide most of the functionality needed.

Selenium WebDriver is one of the most popular web automation tools. It works well with the application when it is run on a local http server but due to the fact that there are not unique URLs for each view, some of the capabilities of Selenium Webdriver are lost. However, it can handle finding elements in the DOM with relative flexibility and ease as

well as sending information to inputs and moving through the application with use of “clicking” which it all that this project needs.

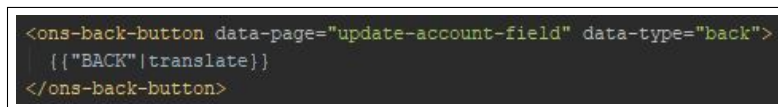
3 Software Design

When developing the testing suite, there were two principles kept in mind in hopes that it would be a useful tool for future developers. The first principle was to make it easy for future developers add new tests. The second principle was to make it easy for future developers to maintain the codebase. This section discusses the development of the testing suite with these principles as guidelines.

3.1 Preparatory Programs

3.1.1 HTML Modifications

By adding custom attributes to the HTML elements in each view, it can provide an easy way for the test runner to determine where in the application it is. Specifically, two custom attributes can be automatically added to each important element found in the application. In this case, important elements are defined as elements that have the ng-click attribute, the ons-back-button tag or the input tag. The first attribute is “data-page” which holds the name of the HTML file it resides on. The idea behind this attribute is to allow the testing program to know where in the application as well as a clear identifier to find all the important elements on the view. The second attribute is “data-type” which denotes which important element it is. Currently there are three types of important elements; a back button, a text box, and elements that move the application to new views.



```
<ons-back-button data-page="update-account-field" data-type="back">
  {"BACK"|translate}}
</ons-back-button>
```

Figure 1. HTML custom attributes

All of these modifications are done at once by simply specifying the root folder containing all the views. When adding new modules to the Opal app, modifying the HTML is as simple as re-running the program.

This program is dependant on JSoup for HTML manipulation of the views. JSoup can be easily added to the project by downloading the library from the JSoup website (<https://jsoup.org/>) and adding the jar file to the build path.

Unfortunately, one attribute, “data-next-page” need to be added manually. (See Section 4)

3.2 Testing Suite

3.2.1 Test Runner

The test runner is the main component of the testing suite. It provides the structure necessary to run through and properly test each view of the application. The test runner has two primary functions: to test list views and to handle the edge cases by routing them to their specific testing functions.

The test runner is based on graph exploration techniques. Through the structure of the list views, the paths of the applications can be modelled as a graph. Each node would be represented as a view while the elements found on the view that move the application to a new view are the edges. With this model, the test runner essentially becomes a heavily modified version of the depth first search graph algorithm. For each list view, we gather the list of clickable elements and click each one and recursively work through all of their child nodes.

Each time the application moves to a new view, the test runner checks if the new view has its own specific function meant for testing. If it does, the test runner routes the program to the appropriate page function. Once it is finished, it returns to the test runner and skips the functionality detailed above.

Finally, adding new test cases to check for in the test runner is very easy to do. Once the new test has been developed, simply add a condition to check for in the routing method and provide the proper function call. The new test case will now be checked for throughout the run of the test.

3.2.2 Page Functions

The Opal Application has many different modules that perform a variety of functions. As a result, several views have to be handled in a specific manner. To organize these tests, each section of the application has its own class containing all the specific tests found in that section as well as an accompanying Javadoc. Edge cases can vary from going through lab results to filling out questionnaires. However, a large share of the views can be split in to 3 general groups based on the view structure and functionality.

The first group is a general list view. In this case, the test can gather all the important elements from the view and test all of them in the same manner as the test runner describes. (See Figure 2)

The second group are the select list views. These views are very similar to the general list views but there are a few important elements that should not be selected. For example, in

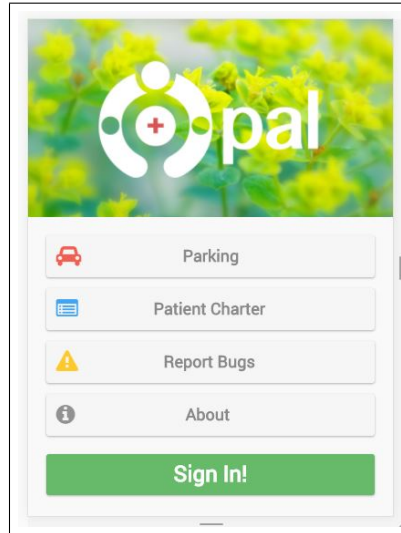


Figure 2. General List: Landing View

the figure below the two list options on the parking view navigate to an outside web-page once clicked. If the test runner were to test all elements, once it clicks the outside link it would break the test. (See Figure 3)

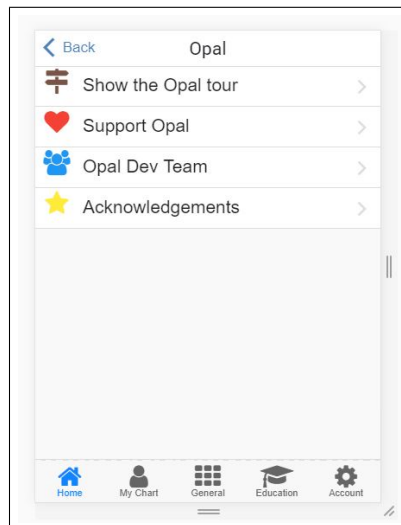


Figure 3. Select List

Finally, there are user input views. This group of views operate in a similar fashion where some text needs to be inputted and then a button must be clicked to submit the information. (See Figure 4)

3.2.3 Individual Page Tester

While the test runner provides an excellent way to test the entire application's functionality, it is not a useful tool when designing new page functions. To provide a streamlined way to develop new tests, a template was developed. Future developers need to do is define

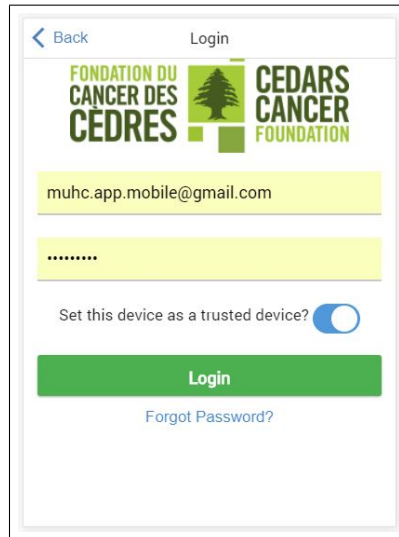


Figure 4. User Input: Login View

the path to the page that they wish to test and begin developing the test in the provided method.

3.3 Executing the test

All of the tests are dependant on Selenium WebDriver to find elements on the view and manipulate them. To use Selenium WebDriver, download the Java iteration from the website (<https://www.seleniumhq.org/>) and add every jar file to the build path.

Executing the test is quite easy. After navigating to the qplus/www folder, execute the command.

```
http-server
```

Once the http server is up and running, check that the IP address list matches the IP address in the test runner and then the test can be run.

4 Limitations and Future Work

4.1 Limitations

The goal of the testing suite was to automate as much as the process as possible, from modifying the HTML to running through the entire OPAL application. It was designed in a way that it can be easily adapted to new modules as they are added to the Opal application. Unfortunately, one aspect of the testing suite is inherently dependant on manual input. For

each element moves to a new view in the application on a click, it requires someone to give it the "data-next-page" attribute. This attribute denotes which view the test runner should expect when once the element is clicked. This limitation should be considered when creating new views as it is easier to do as they are created, rather than all at once.

Another limitation of the testing suite is its speed. Due to the nature of angular based web pages and the speed of Selenium WebDriver, there can be stability issues if they are not properly prepared for. There are times where the element is loaded into the DOM and Selenium WebDriver thinks it is present on the screen but it has still yet to load. As a result, two steps were taken to stabilize the testing suite. Before any operation related to an element takes place, there is a explicit half second wait in order to give the page time to load. After this wait, Selenium WebDriver polls the DOM to see if the element is present in its structure every half second for 10 seconds as a security to make certain the test does not run any instructions before the browser is ready.

Finally, the current design of the testing suite, with its dependence on the HTML tags to give it structure, will not work with the notifications view. The notifications view uses a repeater element which produces elements that move to several different pages throughout the application. The repeater element removes the ability to denote a "data-next-page" attribute for each element. Therefore, the current method of determining what test should be run will not work here.

4.2 Future Work

The future work of the testing suite mainly focuses on maintenance and implementation of new modules.

- Due to lack of time or access, not every view has its own page function to be tested with. For example, the lab results, notifications and calendar modules still need to be developed.
- As more modules are added to the Opal applications, the Page Function classes need to be updated with methods instructing the test runner how to manage each new page added.
- While this testing suite's main focus is testing the structure of the application, it can be used to test content quite easily. For example, adding assertions in a page function can be quite easily done through use of the JUnit testing framework.

References

- [1] A. Kornstadt A. Bruns and D. Wichmann. *Web Application Tests with Selenium*, 26 edition, October 2009. <https://ieeexplore.ieee.org/abstract/document/5222802/>.
- [2] Filippo Ricca and Paolo Tonella. *Analysis and testing of Web applications*, 2001. <https://dl.acm.org/citation.cfm?id=381476>.
- [3] Simon Holm Jensen Anders Møller Shay Artzi, Julian Dolby and Frank Tip. *A framework for automated testing of javascript web applications*, 2011. <https://dl.acm.org/citation.cfm?id=1985871>.

Appendix A: Screenshots

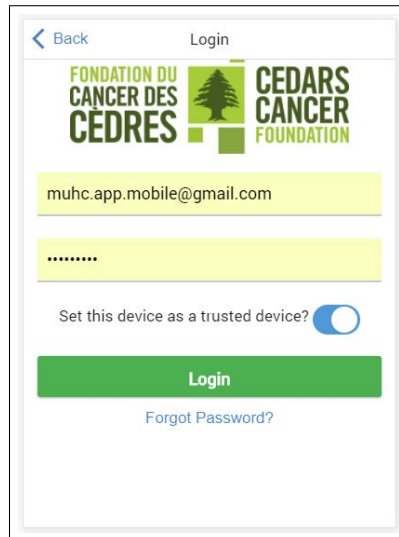


Figure A1. User Input: Login View

```
public static void about(WebDriverWait wait,ChromeDriver driver){
    //selects the opal tour and call the tour method to test it
    skeletonRunner.findElement(By.cssSelector("[data-type='tourModal.show()']")).click();
    tour(wait,driver);

    //clicks the rest of the elements, except the donation link
    List<WebElement> list= skeletonRunner.getElements(By.cssSelector("[data-next-page='content']"));
    for(WebElement element : list){
        skeletonRunner.waitForElementReady(element);
        element.click();
        skeletonRunner.newView("about","content");
    }

    //clicks the back button
    skeletonRunner.findElement(By.cssSelector("[data-type='back'] [data-page='about']")).click();
}
}
```

Figure A2. User Input: Login View

```
<ons-back-button data-page="update-account-field" data-type="back">
  [{"BACK"|translate}]
</ons-back-button>
```

Figure A3. HMTL custom attributes

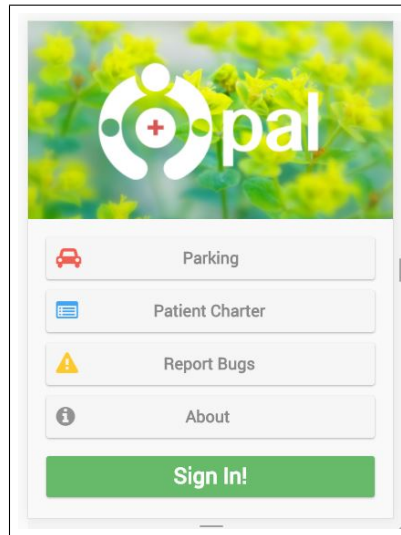


Figure A4. General List: Landing View

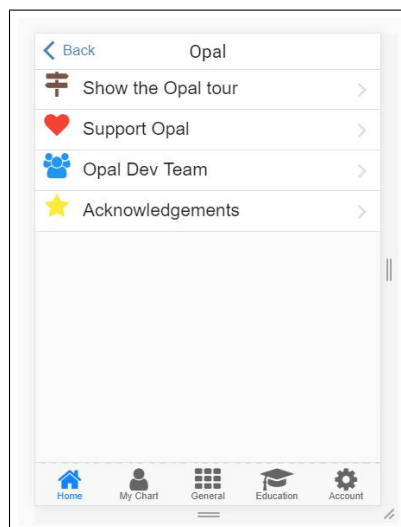


Figure A5. Select List