

Chapter 2

The Test Plan

Introduction

In our exploration in Chapter 1, we learned that Wheelmap utilizes a paid application called BrowserStack to test the JavaScript front-end of the application. It allows for both automated and manual testing of the user interface, checking whether a given sequence of actions by the user produces the expected results. For instance, if a user clicks the filter for “Partially wheelchair accessible,” does the map view change to only mark these locations? Since BrowserStack is a paid application, we weren’t able to view the back-end code driving the tests.

As we began to think about building our own testing framework, Dr. Bowring and Matt suggested the idea of working with Selenium. Selenium is a web-based automation tool for connecting and sending commands to a web browser through Python. After experimenting with it, we have decided that Selenium is a great fit that will allow us to replicate the kinds of tests the Wheelmap team performs through BrowserStack. Our tests will send a set of commands to the browser to navigate around the locally built version of Wheelmap, then report whether they result in the oracle. We list five of the eventual twenty-five test cases below in the **System tests** section.

The testing process/tested items

To devise the major phases of our test plan, we walked through sequences of actions a user might take within the user interface. This brought the following major categories of tests:

1. Opening the application
 - a. Asking user for location access
2. Navigation
 - a. Zooming in and out

3. Filter functionality
 - a. Filtering locations
 - b. Updating the user interface
4. Individual location options
 - a. Adding images
 - b. Opening on separate map applications
 - c. Sharing
 - d. Reporting a problem

Requirements traceability

The following requirements are tested through our first five test cases:

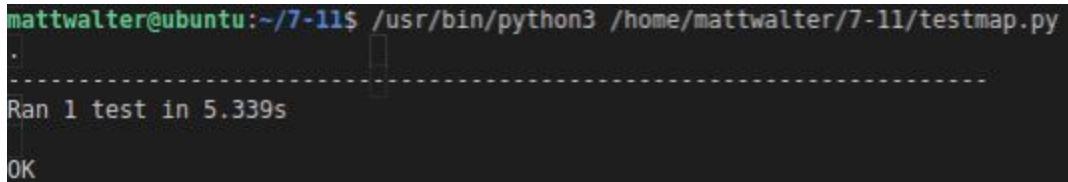
1. First-time users should be prompted with a request for location access so that the app can provide user-specific location functionality.
2. When a user clicks to enable the 'Locate Me' button, the app should respond by highlighting the location icon in blue, indicating the mapview has shifted to the scope of the user's location.
3. When a user clicks to disable the 'Locate Me' button AND the scope of the map view is already focused on the user's location, the app should respond by un-highlighting the location icon.
4. When a user hovers over a button - for example, the 'Only Fully Wheelchair Accessible' - that button should be highlighted in blue to provide an indication to the user of their selection.
5. When a user clicks the 'Only Fully Wheelchair Accessible' button, the app should respond by populating the screen with additional filters such as: 'Shopping', 'Food & Drink', etc..., for better ease of use and efficiency.

Testing schedule

Given the relatively small size of the system and number of test cases, our testing framework will not require scheduling. The idea is that the user can run these test cases at any time.

Test recording procedures

By the direction of the Team Project Specifications document, our testing framework will produce and automatically open a professional-grade testing report detailing the test cases and the results of each test case execution. The Python unittest package produces a report by default after every run of the test cases (Figure 1)

A terminal window with a dark background. The prompt is 'mattwalter@ubuntu:~/7-11\$'. The command entered is '/usr/bin/python3 /home/mattwalter/7-11/testmap.py'. The output shows a dashed line, followed by 'Ran 1 test in 5.339s', and 'OK' at the bottom.

```
mattwalter@ubuntu:~/7-11$ /usr/bin/python3 /home/mattwalter/7-11/testmap.py
.
-----
Ran 1 test in 5.339s
OK
```

Figure 1

This contains the execution time of each test case and whether it passed or failed. We plan to use Python's built-in functionality for writing to and opening files in the implementation. We will include a line within each test case or within method in the driver class (Figure 2) that prints the necessary information to the report.

A code editor window showing Python code. The code imports unittest, selenium.webdriver, and selenium.webdriver.common.keys. It defines a class TestMap(unittest.TestCase) with setUp and tearDown methods. A comment indicates where to insert test cases. The main block calls unittest.main().

```
testmap.py / ...
1 | import unittest
2 | from selenium import webdriver
3 | from selenium.webdriver.common.keys import Keys
4 |
5 | class TestMap(unittest.TestCase):
6 |
7 |     def setUp(self):
8 |         self.driver = webdriver.Firefox()
9 |
10 |    #INSERT EACH TEST CASE HERE
11 |
12 |    def tearDown(self):
13 |        self.driver.close()
14 |
15 | if __name__ == "__main__":
16 |     unittest.main()
17 |
```

Figure 2

Hardware and software requirements

The full procedure to build wheelmap-frontend locally and download the software requirements is outlined in the actual build sequence section of Chapter 1: Deliverable 1.

Short-list of software requirements to build the project:

- Ubuntu Linux virtual machine
- A local clone of the wheelmap-frontend repo
 - <https://github.com/sozialhelden/wheelmap-frontend>
- Node.js 10.x or the latest version
- Node Package Manager (npm)
 - Default package manager for Node.js and the Javascript programming language
- Transifex
 - A web-based translation tool Wheelmap utilizes in order to seamlessly localize the user's experience (i.e. Transifex translates the Wheelmap app for Users based on location)

Procedure to set up software requirements for testing the project:

1. Download python bindings for Selenium:
 - `sudo apt-get install python3-pip python-dev`
 - `pip3 install selenium`
2. Selenium requires a driver to interface with the chosen browser. Download geckodriver for your respective browser. Since we are using Linux systems and they come with Firefox, we defaulted to using that:
 - `wget`
<https://github.com/mozilla/geckodriver/releases/download/v0.27.0/geckodriver-v0.27.0-linux64.tar.gz>
 - `tar -xvzf geckodriver*`
 - `chmod +x geckodriver`
 - `sudo mv geckodriver /usr/local/bin/`

Constraints

Wheelmap utilizes a paid third-party application (BrowserStack) in order to test its software. Since testing is done separately by the developers, we do not have access to the actual test cases they use.

As a result, our testing process is constrained to user acceptance testing. All test cases pivot around ensuring the graphical user interface (GUI) operates as expected when interacted with by the user.

System tests

1. On start up the program will ask for Cookie permissions
 - a. Oracle: popup appears with the text “Okay, let’s go!”

b.

```
testCase1.txt
1  "id": 1,
2  "requirement": elementRendered,
3  "component": CookieButton,
4  "input":
5  elem1 = driver.find_element_by_class_name("button-continue-with-cookies")
6  assert elem1.text == "Okay, let's go!"
7  "output": PASS
8
9
```

2. On start up the program will ask for Cookie permissions
 - a. Oracle: popup appears with the text “Continue without Cookies”

b.

```
testCase2.txt
1  "id": 2,
2  "requirement": "elementRendered",
3  "component": NoCookieButton,
4  "input":
5  elem2 = driver.find_element_by_class_name("button-continue-without-cookies")
6  assert str(elem2.text) not in driver.page_source
7  "output": PASS
8
9
```

3. Tapping the 'locate me' button when active, removes the blue circle from the screen
 - a. Oracle: the leaflet-interactive element is not displayed

```

testCase3.txt
1  "id": 3,
2  "requirement": "userLocationUpdate",
3  "component": leaflet-interactive,
4  "input":
5  elem3 = driver.find_element_by_class_name("leaflet-bar-part")
6  elemDot = driver.find_element_by_class_name("leaflet-interactive")
7  elem3.click()
8  assert str(elemDot) not in driver.page_source
9  "output": PASS

```

b.

4. Tapping the 'locate me' button when inactive enables positioning and adds the user location dot.

a. Oracle: the leaflet-interactive element is displayed

```

testCase4.txt
1  "id": 4,
2  "requirement": "userLocationUpdate",
3  "component": leaflet-interactive,
4  "input":
5  elem4 = driver.find_element_by_class_name("leaflet-bar-part")
6  elemDot2 = driver.find_element_by_class_name("leaflet-interactive")
7  elem4.click()
8  assert str(elemDot) in driver.page_source
9  "output": PASS

```

b.

5. Entering "Halls Chophouse" in the search bar returns the address of the restaurant

a. Oracle: The address of Halls is displayed in the drop down menu as "King Street, 29424"

```

testCase5.txt
1  "id": 5,
2  "requirement": "searchBar",
3  "component": search-input,
4  "input":
5  elem5 = driver.find_element_by_class_name("search-input")
6  elem5.send_keys("Halls Chophouse")
7  elem5.send_keys(Keys.RETURN)
8  assert "King Street, 29424" in driver.page_source
9  "output": PASS

```

b.