SOFT 327
Software Quality Assurance
A2: Test Plan

November 8th 2020

Eric Fillion: 20072951
Cooper Harasyn: 20002241
Jolene Lammers: 20029503
Tingzhou Jia: 20130800

How test cases of different levels are organized?

There are three levels of testing: frontend, backend and integration. For each section, a description of how the test cases are organized was created.

All frontend tests will be stored under a directory called “qa327_test/frontend.” Then, for each component of the system, there will be an individual file that will contain tests for the component. For example, there will be a file under “qa327_test/frontend/test_homepage.py” that will contain the tests for the homepage. These files will contain unit tests in the form of functions using the pytest framework.

Backend tests will be organized in a similar format, except they will be stored under the “qa327_test/backend” directory. Similar to the frontend, there will be dedicated Python files to store tests for each specific section.

All integration tests will be stored under “qa327_test/integration.” Then, there will be dedicated files to store tests for broad functionality that may require multiple pages and both the frontend and backend. For example, there is already a file called “qa327_test/integration/test_registration.py” that tests if the overall system can handle both registration and login at once. Within this file, there is a class called “Registered” that contains three functions: register, login and test_register_login. The first two functions, register and login, perform actions to complete registration and login respectively. The last function, test_register_login, completes both registration and login sequentially to replicate a typical user interaction. The test_register_login function calls the previous two functions to prevent code duplication.

The order of the test cases

The order of the test cases is based on when they occur within a typical user interaction. Cases that appear early on will be tested first, while cases that appear later on will be tested last. For example, for the front-end, the register screen will be tested first since it is the first task users typically interact with, while the 404 error will be tested last since users would rarely go to that page. Of the five requirements that were given for this assignment, here is a list of them in order of when they would appear within testing: login, register, homepage, logout then 404 error.

In addition, the individual test cases within each section will also be broken down in order of when they would typically appear. For instance, testing of the email address format validation would be performed prior to testing the redirect to the homepage upon a successful login.

Techniques and tools for testing

We will be using PyTest, SeleniumBase, and the Python module unittest's mocking functionality in order to test our project. Tests will be grouped into separate directories based on the type of test. For this project, we will be using both unit testing and integration testing, which will be run against every pull request as a form of regression testing.

For the unit tests in our project, we will be using input partition testing. For each unit being tested, we will analyze the functional requirements to create input partitions and create one test for each input partition. This systematic method will allow us to clearly define tests for a given unit, and ensure that each unit can correctly process one example of each different kind of input. We decided to use input partition testing for this project since it will allow us to create tests in a very straightforward and intuitive manner while keeping the number of tests in a reasonable range, as opposed to exhaustive testing, which may exceed our test resources with little value added.

For our integration tests, we will be using functionality testing, where we will enumerate the requirements for the project and test that each of these requirements can be fulfilled. This provides us with a systematic method for deriving integration tests, and creating test cases from the app's requirements will serve to verify the functionality of the app.

We are going to be performing our unit testing and integration testing automatically through the GitHub Actions automated testing flow. Our test cases will be run automatically against every pull request, to ensure that only code that has been thoroughly tested is checked into main. This behaves as a form of regression testing for our project, ensuring that there are no test breakages resulting from new code changes. Depending on the number of testcases and the testing resources available, we may use either full regression testing or selective testing; please view the Budget Management section for more information.

Environments

Our team will be doing local testing using multiple different testing setups. Two teammates are testing on macOS, one is testing using Ubuntu 20.04 under WSL, and the final teammate is using Windows. To minimize environment-related issues, we will be using the same Python packages as specified in the "requirements.txt", and all teammates will be using the SeleniumBase package with the Chrome driver. The teammates own testing environments will allow us to debug and verify locally before pushing code into the GitHub repository.

In addition to the team's local testing, testing will be automated using GitHub Actions. On GitHub Actions, our code will be run on a virtual machine with Ubuntu 18.04.5 LTS. According to GitHub's documentation, this virtual machine is the "Standard_DS2_v2" machine hosted on Microsoft's Azure platform, with two vCPUs, 7GB of RAM available, and 14GB of temporary SSD storage. This should provide ample resources to test our application. Within this virtual machine, we will use Python 3.7 and the same set of Python packages used for testing locally, including SeleniumBase with the Chrome driver, which should closely match our local test setups and eliminate any environment-related issues.

Currently, we are planning to support Google Chrome exclusively in terms of web browsers used to display our app, eliminating the need to test on other browsers.

Responsibility

Test Case Creation and Maintenance

Test cases will be broken down into sections based on requirements. For each requirement/feature we will have one person responsible for building and maintaining all the test cases. Table 1 below shows a breakdown of who is in charge for each test case for the frontend and backend. For integration tests we will also have one person in charge of each section. The person in charge will be one of the people who created test cases for the corresponding frontend and backend sections.

Requirement	**Number of Sub Requirements**	**Who is Responsible**
R1 - /Login	**11**	**Eric**
R2 - /Register	**12**	**Tingzhou**
R3 - /	**10**	**Cooper**
R4 - /sell	**7**	**Eric**
R5 - /update	**7**	**Jolene**
R6 - /buy	**6**	**Jolene**
R7 - /logout	**1**	**Tingzhou**
R8 /*	**1**	**Cooper**

Table 1: Test case responsibility

Who to Contact When a Test Case Fails?

For each feature there will be one person who is the main point of contact if a test case fails, as shown in Table 2. The features will be implemented using pair programming and we will all be responsible for reviewing and approving each code request. However, the person listed is the main point of contact and the main person responsible for the feature. This person may or may not be the same person as the one creating and maintaining the test cases. If the code is failing because of an error in the test cases this person can in turn contact the person who created the test cases.

Feature	**Number of Sub Requirements**	**Main Point of Contact**
R1 - Login	**11**	**Eric**
R2 - Register	**12**	**Tingzhou**
R3 - /	**10**	**Cooper**
R4 - /sell	**7**	**Eric**
R5 - /update	**7**	**Jolene**
R6 - /buy	**6**	**Jolene**
R7 - Logout	**1**	**Jolene**
R8- / *	**1**	**Cooper**

Table 2: Test case point of contact

Budget Management

Due to the nature of this web application, budgeting continuous integration time is required. Each time a pull request is made, a GitHub action will be activated which will launch a VM that will perform tests to verify that modifications did not cause any problems. As the code base for the application grows and more tests are added, there will be a point in time in which running all of the tests will become both time consuming and expensive. So, effective budgeting and monitoring is required to reduce the number of tests that are run while still ensuring that every modification is thoroughly tested.

At first, full regression testing will be used since there will be a minimal number of tests. This will allow the team to focus on developing new features and tests without performing premature optimization. The team will actively monitor the compute time and cost for testing within GitHub's CI interface.

Eventually, the cost of running every test may become too high, at which point the team will begin to use selective testing. The programmer will specify which set of tests to run when they create a pull request. For example, if the programer modified the login page, then they will specify for GitHub Actions to run the login page tests. The cost of running each section will be monitored and the programmer will determine which sections they believe will be the most beneficial to test for each pull request while considering the cost of the section.

However, selective testing is not perfect and may result in some errors going unnoticed. For example, a modification to the login page may in some way impact the homepage, but with selective testing the homepage tests may not be performed. To address this shortcoming, prior to releases, exhaustive testing will be used for each section of the codebase prior to the code being released to production.