Benjamin Lang; Amrit Kooner; Leo Belanger
Ryan Liu; Andy Wong; John Sun

SmartShop

28th October 2016

TEST PLAN DOCUMENT

Introduction

Our group plans to create a user-friendly Android and web-based application that allows users to create shopping lists and plan shopping routes based on prices and/or proximity of stores. It will also provide user profiles and recommended items to the users. This document outlines the testing plan that will be carried out to ensure that our product (SmartShop) is of sufficient quality and meets the needs of our users.

Our product revolves around Android and Web for user interface (UI) and an external server providing multiple services such as database operations, data retrieval via crawlers and client handling. Testing communications between these modules will be vital to producing a high quality product that is both intuitive to the user and easy to use. In terms of functionality, the most obvious testing revolves around modules building and handling correct messaging objects (through JSON) as this is critical in communication and the system would fail if our software is not correct. Further testing strategy and some of our planned test cases will be explained below.

Verification Strategy

We plan on verifying the suitability of our application by obtaining weekly feedback from our peers, family, as well as professors. Initially, we will show them mock-ups of the the web site and Android UIs (with limited functionality) and ask for their opinions regarding the overall appearance of the UIs (i.e. is it pleasing to the eye?). In addition, we will ask for suggestions on how the UIs could be improved, such as different placement of UI components (or addition/removal of UI components), different colour themes, and different styles. The UIs will be continuously updated throughout the project as the application's functionality is developed and the users get a better idea of what features they would like in the UIs.

In terms of verifying the core functionality of the application, we will show them a working demo of the application with a small set of searchable items. They will then be able to search for these items and create a shopping route that is optimized based on geographical proximity and/or price range. We can then ask for their opinions on some of the following topics:

1. How easy it is to understand the application via its UI? (e.g. searching for an item, creating a shopping list, making an account)
2. The responsiveness of the UI (is it "fast" enough?)
3. The quality of the application's functionalities, such as:
    a. The relevance/accuracy of search results
    b. The relevance/accuracy of recommended items to shop for
    c. The degree of optimization in the shopping route (in terms of time and budget)

We will also ask them for suggestions on how existing features should be modified or which additional features should be added to better meet their needs as consumers.

If a working demo of the entire application is not ready, then individual aspects of the application can be shown and verified. For example, even if accounts cannot currently be created, we can show the item recommendation feature by allowing the user to enter multiple item searches and then use those searches to display a selection of recommended items to shop for.

Non-Functional Testing and Results

Our first main non-functional requirement is performance and in particular how responsive the UI is. To assert a performance standard in our product, we will utilize the locust open source load testing tool to develop a test suite that will track the responsiveness from user input to result. As an example, we will develop mock clients to query the database for one or more items of varying types and visualize the results with respect to query size and query types. Furthermore, this will be replicated over multiple connection types such as home, enterprise and mobile data connections.

In addition to UI responsiveness, system stress testing will be used to examine our system under high user loads in order for our team to see at what critical load the system starts to experience a large decline in overall performance. Locust allows us to create multiple concurrent test clients to overwhelm our service and we will able to plot a response time vs. concurrent users graph to rate our system's stability. Using this information, we will be able to locate the bottlenecks in our system and optimize the slower operations in the system.

The other concern was the security of our system. Multiple connections exist, the most prominent being the connection between the Android/Website UIs and our main server that connects to the database. Because our application stores extremely sensitive information such as username, email, as well as user location, it is critical for us to make sure there is no security flaw within our system. In regards to the testing, we plan to use different software readily available online to test the security of our program. Such programs include Google's Skipfish, detectify, soapUI, Scan my server, SUCURI and other programs. These programs are used by industry professionals in order to verify the security of their websites.

An example is using soapUI, which is an extensive testing program used by many big companies such as Apple and mastercard. SoapUI allows us to detect security flaws inside programs that rely on application layer protocols like HTTP and TLS/SSL, which is our primary method of transferring secure data. Another good program we can take advantage of is detectify, which analyzes a website from a hacker's perspective and tries to find openings in our security to gain access to sensitive information. In fact, most security programs focus on testing security through NoSQL injection, XSS (passing of malicious JS), Server/Client authentication and authorization, password hacking, session timeout, and many other forms of attacking a web server. We plan to use multiple different programs to fully test and patch all security flaws.

Benjamin Lang; Amrit Kooner; Leo Belanger
Ryan Liu; Andy Wong; John Sun

Functional Testing Strategy

We plan on running weekly integration tests so that we can quickly spot bugs in newer versions of our code and be able to revert to working iterations while the bugs are fixed. For example, if a newer iteration of the crawler incorrectly adds data to the database, we can revert the crawler to last week's iteration while the bug is fixed; this way, the rest of the system still functions correctly. The reason we chose weekly integration tests is because daily integration tests will be too extensive as generally we will not have major code changes being done daily, but rather over the course of about a week.

For the backend logic of the Android application, main server, and the website, we will be testing them using unit tests. The aim is to test the correct implementation of classes, so we will be testing after the completion of each class. The test cases will be based on the requirements for each of the functions, with the inputs being based on the format of real data, and the output being what we expect each of the methods within a class to perform.

In terms of handling bugs, we plan on using a less formal method such as a spreadsheet because there is less overhead in getting it set up and it is always easy to change the layout of the tables. We already have a group folder and it will be easy to add additional documents and bug tracking reports into it. We will use categories consisting of:

- "Severe", meaning it directly affects behaviour and has top priority
- "Moderate", meaning it might not directly affect behaviour but could in the future if not addressed
- "Minor", meaning it has a small impact on performance but major functionality still works

Status of bugs will include "fixed" to indicate the bug has been fixed but needs to be further verified, "closed" to indicate the bug is fixed and verified, and "open" to indicate the bug has not been addressed yet.

Adequacy Criterion

We will make sure that at least one test exists for each requirement and for each use case. We chose this over the other criteria because if our requirements and use case tests function correctly, then we can be sure our smaller methods must also function correctly. On the other hand, if the requirements and use case tests do not function correctly, then we can write more extensive tests to individually test each method and locate the bugs.

Benjamin Lang; Amrit Kooner; Leo Belanger
Ryan Liu; Andy Wong; John Sun

Test Cases and Results

Test #	Requirement Purpose	Action / Input	Expected Result	Actual Result	P/F	Notes
1.	Accuracy of Crawled Data	Randomly verifying entries in the crawled JSON documents.	The fields such as image and price for the randomly chosen entries agree with the actual image and price on the website catalog.			
2.	Responsiveness of UI (Platform)	Client requests from website and different Android devices.	We strive for a response time of less than 5 seconds for a query size of 10 items. This should be constant across multiple Android devices.			
3.	Responsiveness of UI (connection type)	Client requests from website and different Android devices.	Response times may vary due to better or worse connections with mobile data being the slowest, followed by home connections and then enterprise connections.			
4.	Server stress test	Multiple concurrent client connections using Locust load testing tool	Ideally we would like to be able to have ~750 semi-concurrent connections to be the threshold before we decide that a large downgrade in our service performance is considered acceptable.			
5.	Security Test #1: Brute forcing passwords	Running a script to brute force a user's password	Request timeout - Web server will recognize malicious attempt and block out that user from sending too many requests (5) within a short time frame. Will also include CAPTCHA to block out bots.			
6.	Security Test #2: Using detectify and other programs	Running such programs on our web server and starting a test for security flaws.	Our website will have no security flaws from which hackers will be able to retrieve sensitive information. In addition, no malicious scripts can be ran on our website to possibly infect end-users.			
7.	Formatting of JSON Messages	Running a script to verify that received messages are properly JSON formatted.	The script will indicate that the messages are properly JSON formatted. If they are not formatted correctly, we will need to revise how the messages are sent.			

8.	Verification of Server Responses	Running a script to verify that the server sends correct responses based on messages verified by Test #7	The script will verify that correct responses are sent back by the server, assuming that the requests are properly JSON formatted (Test #7).			
9.	Verification of custom view adapter	Running unit tests to verify that the adapter retains the correct information and displays them properly.	The JUnit test will pass the adapter trivial objects and attempt to retrieve them, which should return the same objects that were passed. Then the returned view should be matched with the correct one which the test specifies.			
10.	Validation of UI interactions	Running unit tests to simulate the user accessing all major UI elements and validate the action.	The JUnit test will check for changes whenever a UI element is accessed, and we are expecting that the elements have changed to the expected/correct states as specified in the test.			