Testing Plan Ver. 1.1

April 5, 2018

Team Name:
Gnosis Solutions

Team Members:
Kalen Wood-Wardlow, Christopher Simcox, Thomas Back, Kristoffer Schindele

Project Sponsor:
Dr. Leverington

Faculty Mentor:
Dr. Leverington

Table of Contents:

Introduction

Grocery shopping chains in the past few decades have offered rewards programs that allow customers to follow daily deals and shop wisely, while the store collects data about customer shopping habits. Unfortunately, there are rarely any special programs that allow the customer to benefit from collected data such as: what certain items are in stock and helpful details about those items like seasonal availability. With the exception of curbside pickup or delivery programs, there are rarely any tools for the customer that will allow them to plan and shop more effectively. For this reason, customers develop an affinity for a “familiar” store that they visit exclusively to cut down on time spent looking for items or sales. This is done without any consideration that there may be stores available in closer proximity to their location or needs.

Our motivation for testing involve really honing in on a simple to use application with a beautiful UI that users will enjoy looking at while using our app, even keeping them using our app longer to due the the aesthetics of the application. Some other features that need testing is the scalability of our database, along with the ability to use geolocation.

Our overall testing plan goes as follows:

- User Experience (UX) Testing *
- Module Unit Testing
- Performance Testing *
- Integration Testing
- Usability Testing
- User Interface (UI) Testing *

* Emphasis on these tests

The reason we are using this testing plan as this allows us to fine tune the app by having our client, and other users really getting to know our app which allows them to provide feedback to us that provides a quality way to improve our application features and UI. Which in turn will give us a simple to use app with an elegant UI that users won’t want to put down.

Unit Testing

This project requires not only that all functionality be present from our requirements specification but also great performance. For this section of the document it is prudent to talk about both functionality and performance testing. Unit testing is where you test individual modules, or units, of code to make sure they are operating correctly. The first item to look into is the testing of individual module functionality.

Module Unit Testing

In this project there is a lot of functionality which are developed into each module. For the testing of this software it is important to test each module by itself and in the system to make sure that the software will not cause errors in future prolonged use. The modules that will be tested are Geolocation, Price comparison, Store recommendation, Database controller, Screens, and Integrated packages controller.

The Geolocation unit testing will be performed on both getLatitude() and getLongitude(). Both will check to make sure that it always returns a valid output or a checkable error code when necessary. This is very important for almost all of the services this application offers. Since this is very important there should be a zero tolerance for failure. The system should be robust and should simulate as many locations as possible. Another point of testing is to see how accurate the geolocation is in terms of feet and miles to make sure the system can be calibrated. For most purposes 3 miles is enough accuracy, but in other cases if a user is in a particular store then it would be desirable to make it as accurate as the possible store size.

Next up is the price comparison which is how we offer lower price recommendations close to the user. While the price comparison module relies on geolocation to find stores close to the user price comparison still has important functionality to be testing independently. To test for this the functions that process the database items will be given a sample set of real past database items used in a previous run of the application to make sure it returns the correct values such as the correct lowest price from a location that is close to the user location given. The guidelines on this is that it should always return the lowest price of the stores that are evaluated and picked using the store recommendations module as explained later.

The store recommendations module allows us to find the closest stores to the user to show prices of items so we are not recommending prices that are far away in terms of distance/travel time. By this we will create some sample geolocational data that is possible to look on at a map and determine by hand what is the closest stores and then the test would input that data into the functions of store recommendations module and see if the same store(s) are found as closest.

Database controller testing is very easy since the way the database was setup in the project was relatively simple. This type of testing would be to verify that the functions pull the correct data even through exhaustion or excessive pulls/accesses of the database. The controller encompasses all message passing to the database. The testing guidelines on this is that there should be absolutely no way to access the data outside of this function unless the caller is the application itself or the database. This will be measured by unit tests trying to mimic a web call from a non mobile platform. Another guideline is that the database message passing must always work. This will be tested by creating simulated data that is both invalid and valid to make sure error handling as well as normal functionality is well tested. Finally, lets talk about the one module not listed so far and that is the Application Programming Interface (API) library module.

There is however, one module that will not be tested. This is the API library module which houses all of the API calls to external services that was developed by the providers of the services in use such as Facebook and Google SignIn and Firebase. While some of these calls were used in other modules such as the database controller the reliability is with the service hosting companies.

Next it is important to address the testing of performance. There is not a need to write test code for performance as both Android and iOS platforms provide performance testing software in their own respective IDE's for use.

Performance Testing

As shown in figure 1 below there are advanced tools for memory and computational impact/usage that iOS gives us. While Figure 2 shows how Android provides live usage information.

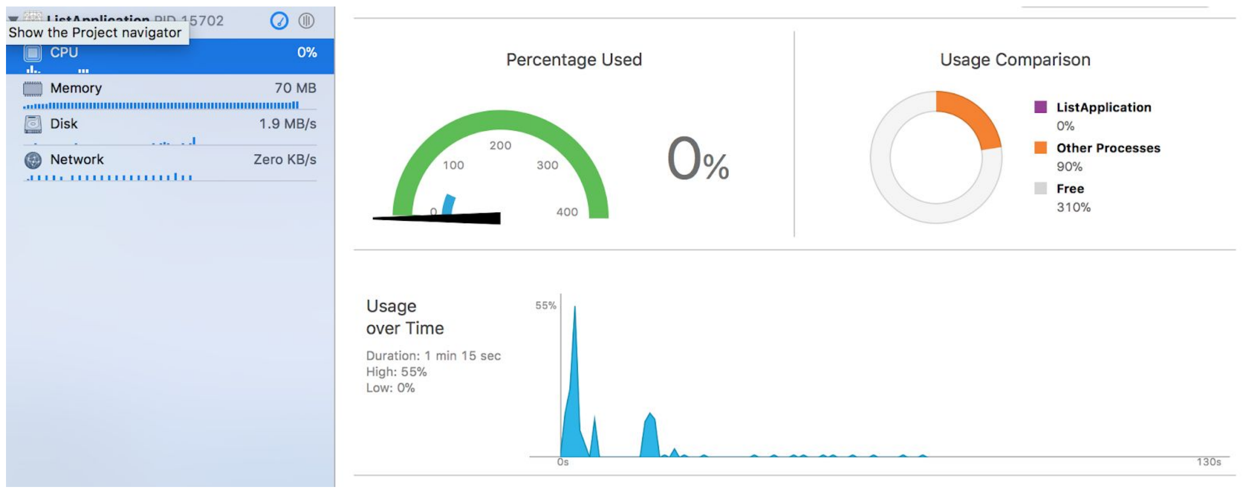

Figure 1

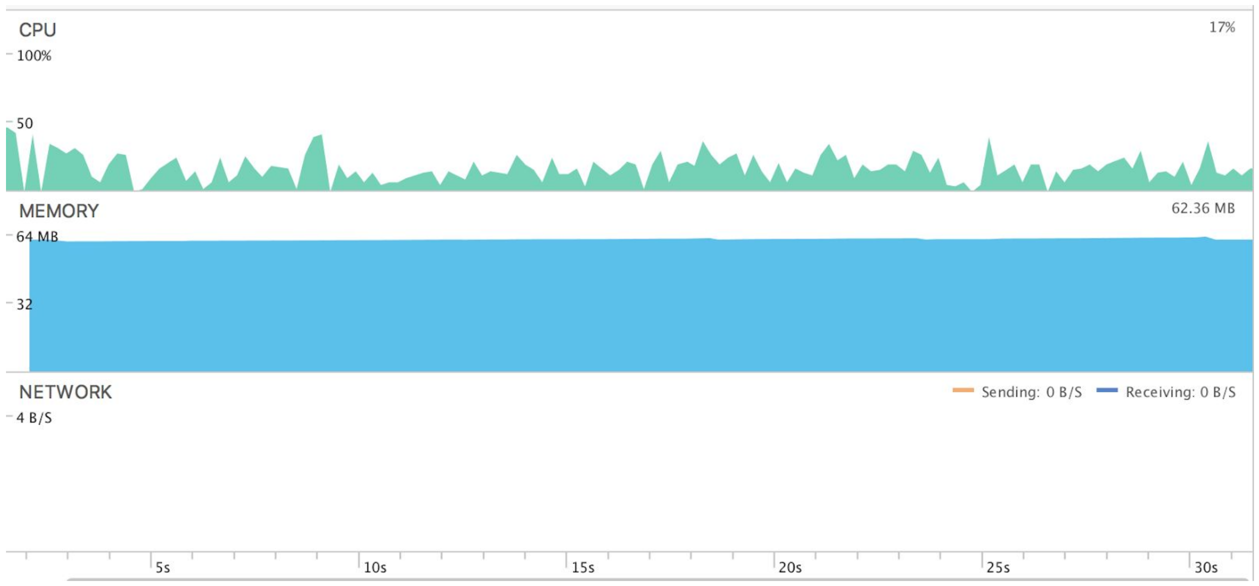

Figure 2

These screenshots were taken from real time input given from both Swift and Android Studio Interactive Development Environment(IDE). When needed, live inputs for these performance statistics can be provided from a live demo with any client requesting such detail for verification purposes. However, using this information panel while running the application may inhibit or reduce the performance of the system. The agreed upon guidelines were that the application is not to use more than 10% of Central Processing Unit(CPU) during normal operation after startup and while in background. The system must also not use more than 1MB per minute on download and uploads to conserve battery by way of the network card activity.

With this information from performance and modular unit tests it is believed that the application will perform to the specification outlined in the original requirements document. Next, it is important to go into the integration testing which encompasses testing of the system as a whole with all subcomponents integrated.

Integration Testing

The application has different modules that all tackle a specific acquirement and handling of data, such as our geolocation and database modules. The appeal of the application stems from its ability to integrate location, user data, and services handling user specific data alongside database information, into one seamless and convenient system. Therefore a testing plan to ensure proper communication and handling between these modules is essential. Key integration points within the code will be chosen and requirements for each will be set in this section.

The first and foremost component is the data entered by the user. This must be entered locally and stored in the cloud database under the correct fields. When creating a list, that list must only show up under that user's table, and the items in that list should only appear under that list's table (as well as copied under the stores table when the user enters prices while in shopping mode). Unit tests can be created in which getter functions can retrieve a desired data value from the database after it is stored. This test will use the same value to procedurally upload the data, such as "apple" as an item, and then run the getter function to fetch and check the value of the submitted item from the database and check that these values match. This method will be appropriate as the retrieval of the data will be directly from the database, and will be compared with the data submitted from the user level. The same tests can be applied to separate lists to ensure an item entry was not placed in the wrong location. The unit test could be designed as follows: a script written to invoke the addNote() method on a new list with the a global string value "Orange" as the input. After this the GetAllItems method would be called expecting a value of "Orange" as the only array element. This test would ensure item values get added properly, and the same unit test could be modified for a new list itself, instead looking for a return of the correct list name's string value, as well as returning the path to ensure its under the correct user. Modifying list items will follow the same structure. Editing an items name or price would require the database value requested be equal to the new target, defined beforehand. Deletion testing will also return the item array and check that the deleted value is nonexistent. Again, the functionality to return paths to certain levels of the database already exists meaning those functions could be called again to check whether these items are not in other locations, but it can be assumed that the correct path returning the right item means that the endpoint is the only location it has been submitted, as the database functions submit to a single path, but this could be implemented if the concern were to arise.

A second major component of the application is the geolocation technology. This exists as a separate API outside of react's main library. This API also harnesses and communicates with the phone's GPS hardware, giving more room for error than would exist simply through software communicating. The geolocation functionality set up for now are two functions, one returning a longitude, and one returning a latitude. These are submitted to the database as a store entry, with the store title, and the longitude and latitude defining it. The process is initiated by a store submission call from the user level, which executes the geolocation API, returns the longitude and latitude, and finally submits these to the database. In this situation confirming that the database received the right coordinates will ensure that data was passed properly along the route. The accuracy of the coordinates in question will be handled by the unit testing proposed in the above section. As with the tests written to ensure reliability of the user data into the database, these tests will be similar where the latitude and longitude returned immediately at a given location by the geolocation functions, will be stored, and then checked against the values sent to the database by retrieving from the intended path. If these are correct the test will pass.

The price location feature will also require correct loading into locality the prices from correct stores. This testing may need to be done with a human observer who will check the price values loaded into the comparison screens against the ones visible from firebase's console. This is proposed since the getter functions to load the prices into the comparison console are the same that will be used to fetch them to check within the test's scope, making them from the same location. Using these methods would not allow any disruption in the path to be seen. If a tester can observe the correct values are returned for a given store path from the console, then the test will be a success. The following steps focus on acquiring feedback for the user experience through usage of these components in the user interface.

Usability Testing

The final and most important aspect of the testing plan is Usability testing. Usability testing encompasses all aspects of the user interface that provide features or functionality to the user. In the case of this application, Usability testing will be performed through four distinct methods: user interface (referred to as UI) survey testing, blind user interaction testing, user collaboration testing, and user expectation testing. The following section details how each of these testing methods will be carried out as well as what elements of the application's development that they contribute to.

UI survey testing focuses on exposing beta-testers to similar UI layouts, and subsequently collecting subjective evaluations from those beta testers in an effort to quickly determine optimal configurations of the user interface. The similar layouts will be generated by making small directed changes in details such as: font size, icon size, the arrangement of items in the interface, coloration of interface items, and sensitivity of swipeable interface items. In order to isolate the evaluation of each detail, similar configurations will only have one altered element. Each configuration will be exposed to at least 10 users, with most of them only requiring a single test containing at least three choices. Tests with inconclusive results will prompt additional tests with fewer options to narrow in on a prefered configuration.

For example: three interfaces are generated all with the same source code except for that one uses size 12 font, the second uses size 14 font, and the third uses size 16 font. These three interfaces are then presented to at least as many beta testers as there are options and accompanied with evaluation questions such as: ‘Which layout do you feel is easiest to read?’.

The tests will be issued via paper handouts containing relevant graphics, and the written responses will be collected after the testers have made their decisions. Following data collection, the results will be interpreted to determine whether a final decision can be made, or if additional rounds of survey should be conducted. Layouts focusing on items that are purely aesthetic can be presented as sets of screenshots, while items that require user interaction to evaluate will be presented through interactive demonstrations. This type of testing focuses on the minute details of the UI, and thus will be the last type of testing during the general user testing period.

Blind user interaction testing aims to gather data on the usability of the system as a whole. Users who have never seen the application before will be invited to sit down for a few minutes, attempt to use the application without any instruction, and finally answer a few questions about their experience. The results from these tests should reveal glaring issues in the user’s process of learning the application. Since the app is meant to be extremely easy to use, it’s vital that users are able to intuitively understand the interface with minimal to no instruction. In contrast to the directed questions in survey testing, interaction testing questions will be more open ended in an effort to motivate the testers to critically evaluate their own experience. Blind user interaction testing will be performed with at least 10 different subjects and each will have between three and five minutes to explore the app.a

For example: ‘What was your favorite part of the user interface, and why?’ or ‘Which part(s) of the app did you have difficulty figuring out how use? In what ways could the app be altered to avoid that difficulty?’

Due to the importance of the user’s ease in learning the application, this will be the first type of testing performed, as identified issues could lead to drastic changes in UI design.

Collaboration testing focuses specifically on improving the value of the application’s crowdsourcing element through longer-term tests carried out by beta testers. Groups of users no smaller than two will be given access to the application and instructed to use it the next few times they go shopping. During this period, beta testers will be directed to shop at least one store they frequent, and one that they have either never been to or are only partially familiar with. At the end of the testing period of no less than seven days or two shopping trips, the data users entered will be collected to evaluate a number of the application’s features. These include: data duplication detection, the number of items typically entered for a list, and how often a collaborating user accessed data entered by one of their peers. While motivating users to enter information is not a requirement for this project, these tests can still be leveraged to make the crowdsourcing process more efficient by eliminating obstructions or difficulties collaborating users experience while entering or accessing shared data.

Collaboration testing will be performed shortly after and concurrently with blind user interaction testing, as the crowdsourcing aspect of the app's usage is critical to fulfilling the project requirements. Due to the complex and time intensive nature of collaboration testing, only three groups of users will be required for our tests. However, more tests will be run in the event that more beta testers are available for such an extended period of time.

Finally, user expectation testing will focus on identifying common user expectations of the app's performance and resource consumption. Similar to blind user experience testing, testers will be prompted to sit down with the app and perform a comprehensive list of actions. These will include actions such as: creating a list, adding items to a list, editing/deleting items, changing user settings, and using the app in both online and offline mode. The demo of the app will be followed with a questionnaire that focuses on gathering the user's qualitative analysis of the app's performance. This type of testing will be used to optimize the user's experience through eliminating performance obstructions to their experience.

For example: 'Did the app feel responsive to your actions, yes / no? Based on your experience, please rate the response speed of the app's interface on a scale of 1 (slow) to 5 (fast).'

As the user's expectations of the app's performance are important, but less influential than the design of the user interface, expectation testing will be conducted following completion of blind user interaction testing and concurrently with collaboration testing. Ideally, expectation testing will be performed on the same group of at least 10 users that blind user interaction testing is performed on.