The University of British Columbia

L o c a l l y

Test Plan Document

Angy Chung

Anna Gudimova

David Jung

Mo Kiani

Andy Lin

Alena Safina

October 28, 2016

Introduction

Locally is an Android application that will allow users to view locally grown produce being sold at farmer's markets, check what's in season, and view nearby vendors. The data will come directly from the vendors themselves as they can update their daily produce stock through our application.
The application back end is hosted on AWS DynamoDB storing tables with app data in a non-relational database. The Android application backend handles user interface and requests to the database and is written in Java. The Android application UI design is written in XML following Google's Material Design guidelines.

1 Software Verification

The users of our application are vendors as well as consumers. In order to ensure that the product meets the real needs of our users we are working with actual vendors, notably members of the UBC Farm. From them we are obtaining information on what they would like to see in our application and areas that need our focus. At the very beginning on our project, we compiled a list of questions to ask vendors and we approached several vendors on two separate occasions, including members of the UBC Farm as well as vendors from Mt. Pleasant community farm, explaining our idea and asking them these questions. From them, we extracted information such as whether or not they were willing to put in the time to set up a vendor profile, what information they would like to include on their vendor profile pages (photos, open or closing soon status etc), as well as what common questions they got asked by customers and how our application could help answer these consumer's concerns.

2 Software Validation

We will be ensuring our implementation meets both functional and non-functional requirements in various ways, using unit tests, integration tests as well as system tests to test the software we produce. We will be focusing first on testing high priority functional requirements such as CRUD operations on vendor data, user searches on produce and vendors, as well as fetching market data and displaying data on our map and calendar. Non-functional requirements such as ease of use, and performance will also be tested accordingly, with these test results documented and compared with the objectives we have outlined below.

2.1 Testing Functional Requirements

REF #:	**Functional Requirement**	**Objectives**	**Priority**
F1	**Fetch Market Data**	View markets in multiple ways ● As a list ● On a map	High

		● On a calendar View attributes of the market ● Hours ● Dates ● Locations ● Vendor listing	
F2	**CRUD Vendor Capabilities**	Vendors must be allowed to define their own attributes and have customers see changes in real time ● Name ● Description ● Product List ● Photo ● Produce in Stock	High
F3	**Fetch Item**	Allow users to search for a given item Allow vendors to stock a given item	High

2.1.1 Local unit tests and instrumented tests

Android tests are based on JUnit and can be classified as ***local unit tests*** and ***instrumented unit tests***. The test code must go into one of two different code directories depending on the test type.

Local unit tests

Located at module-name/src/test/java/. These tests run on the local JVM and do not have access to functional Android framework APIs. Unit tests are fundamental tests that exercise the functionality of the smallest possible unit of code (which could be a method, class, or component) in a repeatable way. We will build unit tests to verify the logic of specific code in our app.

Local tests run much faster compared to the time required to deploy and run the test on an Android device. By creating and running unit tests against the code we will verify that the logic of individual units is correct. We will run unit tests after every build to quickly catch and fix software regression bugs introduced by code changes.

Instrumented unit tests

Located at module-name/src/androidTest/java/. Instrumented unit tests are unit tests that run on Android devices and emulators instead of running on the Java virtual machine. These tests have access to the real device and its resources and are useful to unit test functionality which cannot be easily mocked by mocking frameworks. An instrumentation-based test class allows tests to control the life cycle and user interaction events, ex. send key/button events (or touch events) to the application under test.

For now we plan to take advantage of the Testing Support Library which provides APIs that will allow to quickly build and run instrumented test code for our app. Library includes a JUnit 4 test runner (AndroidJUnitRunner) and APIs for functional UI tests (Espresso and UI Automator).

User Interface Tests

UI tests are a subclass of instrumented tests, located in the same directory. User interface testing will help us to ensure that users do not encounter unexpected results or have a poor experience when interacting with Locally. The naive approach to UI testing is to have a human tester perform a set of user operations on the app and verify that it is behaving correctly. However, this manual approach can be time-consuming, tedious, and error-prone. We chose an automated approach that will allow us to run tests quickly and reliably in a repeatable manner.

The Espresso testing framework we will use provides APIs for writing UI tests to simulate user interactions on the target app, in order to perform testing tasks that cover specific usage scenarios. A key benefit of using Espresso is that it provides automatic synchronization of test actions with the UI of the app being tested.

2.1.2 Integration testing

Integration testing will be done on each branch merge following an independent unit testing done prior to merging. This will ensure that new features/interfaces being merged into the main project are bug free and any regression bugs that appeared due to the new merge are spotted.

Same tests: local unit tests and instrumented tests will be written for integrated modules to test the adequate interaction of merged components, with the use of same tools and libraries for automated testing.

2.1.3 System testing

System testing will be done for the code on master branch that should ideally be a working copy of the project, or have most functional features integrated. We will write test cases based on the use cases in our requirements document and the common use scenarios (see chapter 4 for tests description).

Since the vendors are the only users allowed to create a profile and enter data like vendor name, stock items, location, password, and attach vendor image, those entries will be the ones we will test for invalid inputs. Possible invalid inputs can be empty strings, non ascii characters, special characters and punctuation signs, invalid password length/strength/characters, and many more that we will describe in the test table in the end of the document.

2.2 Testing Non-Functional Requirements

AWS DynamoDB platform was chosen to avoid issues with load balancing, scaling, and handling credentials, as we mentioned in the design document. Also, since the app covers market events only in the Vancouver area (having 8 farmers markets currently) we

do not expect a high volume of usage in the nearest future. Therefore, we will not consider any scalability issues with the app for now. As for maintainability, our database can easily have more vendors/markets entries inserted, UI objects for which will be created automatically according to the template as long as we fill in all the information fields for each entry (ex. market name, location, dates, and days for a new market entry). See the table below for objectives on the other two non-functional requirements: performance and ease of use.

REF #:	Structural Requirement	Objectives	Priority
NF1	**Ease of Use**	The system must be user friendly. The user interface will have: ● Meaningful field names ● User prompts/hints ● Multiple ways of doing one thing eg. Viewing markets ● Links to other features on the page	High
NF2	**Performance**	● App response time (in response to new user input) should be 100 to 200 ms for users to perceive it properly functioning ● If app is doing time consuming task (e.g. loading a map) show that progress is being made (e.g. fill the information asynchronously) ● Use performance tools such as Systrace and Traceview to determine bottlenecks in app's responsiveness.	High

3 Adequacy Criterion

We need to define at what point we are confident that the project is completed, dependable, and hence can be launched.

Adequacy criteria are fulfilled if there is an adequate set of test suites that are passed by the system. These test suites in turn must have sufficient test coverage, meaning that for each use case and each requirement listed, there is at least one test that covers it. The test suite not only needs to cover every aspect of code but also ensure its functionality is consistent with the specifications written for that use case.

For greater rigour, it is important to make sure that each method written is tested.

When testing our app we shall make sure that all lines in our program code are executed at least once in our tests. With the new Android Studio, we will be able to run our unit tests and see the coverage all within the IDE.

4 Test Cases and Results

Test #	Requirement Purpose	Action / Input	Expected Result	Actual Result	P/F	Notes
T1	Vendor Search	User presses on search icon and types in vendor name	A list of vendors that match the vendor name start letters shows up	TBA	TBA	
T2	Vendor Login	User enters email and password combination in the corresponding fields	Vendor profile page shows up if email and password combination is correct, otherwise, error message is displayed	TBA	TBA	
T3	Load Vendor Profile Page	User selects a vendor from the displayed vendor list	Vendor profile page shows up, indicating information about the selected vendor	TBA	TBA	
T4*	Read market location data	User selects the map option from the navigation drawer	Map fragment replaces main content and drops pins	TBA	TBA	
T5	Update daily stock	Vendor types in the name of the new stock item and presses the add stock button	Vendor profile is updated with the new stock item, the new stock item is added to the vendor's table entry in the database	TBA	TBA	
T6	Load Market Description Page	User selects a market from the displayed market selection	A page containing market description, location, times of work and directions is displayed	TBA	TBA	

*A more detailed test case description of test number T4 is available below. Ideally, if we were given more time, we would like to write out detailed test cases descriptions for every single test case. The tables below provide an idea of the level of detail we would like to include, using test case T4 as an example.

Test Case Information

Test Case ID:	**T4**	**Version:**	**1.0**	**Version Date:**	**10/27/16**
Test Case Title:	Read market location data				
Project Name:	Locally				
Use Cases:	UC001: Map Fragment Screen				

Test Case Summary

Test Case Description:	Fetch market data from the database
Test Case Objectives:	Display the location of the databases on Google Maps
Measures of Success:	All locations shown on the map
Target test date:	TBD

Test Case Sequencing

Test Case Predecessor:	N/A
Test Case Entry Criteria:	1. Previously connected to the Internet in application 2. On main activity and click view on map in drawer item
Test Case Exit Criteria:	1. Error dialog 2. Maps updated with pins