The University of British Columbia

L o c a l l y

Design Document

Angy Chung

Anna Gudimova

David Jung

Andy Lin

Alena Safina

October 17, 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.

This documents explains architecture and design of the system to be built.

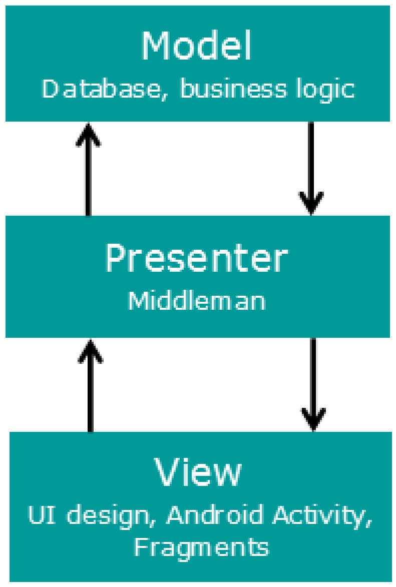
SYSTEM ARCHITECTURE

The application consists of the following major components:

1. Non relational database (specifically AWS DynamoDB) that will store tables with application data with different access permissions for consumers and vendors.
2. Java for the Android application backend to handle user interface and requests to the database
3. XML for the frontend Android application UI design following Google's Material Design guidelines
4. MVP design pattern

Model-View-Presenter Architecture

The model-view-presenter (MVP) architecture pattern is a pattern adapted from the of the model-view-controller (MVC) pattern. The components of the MVP pattern are:



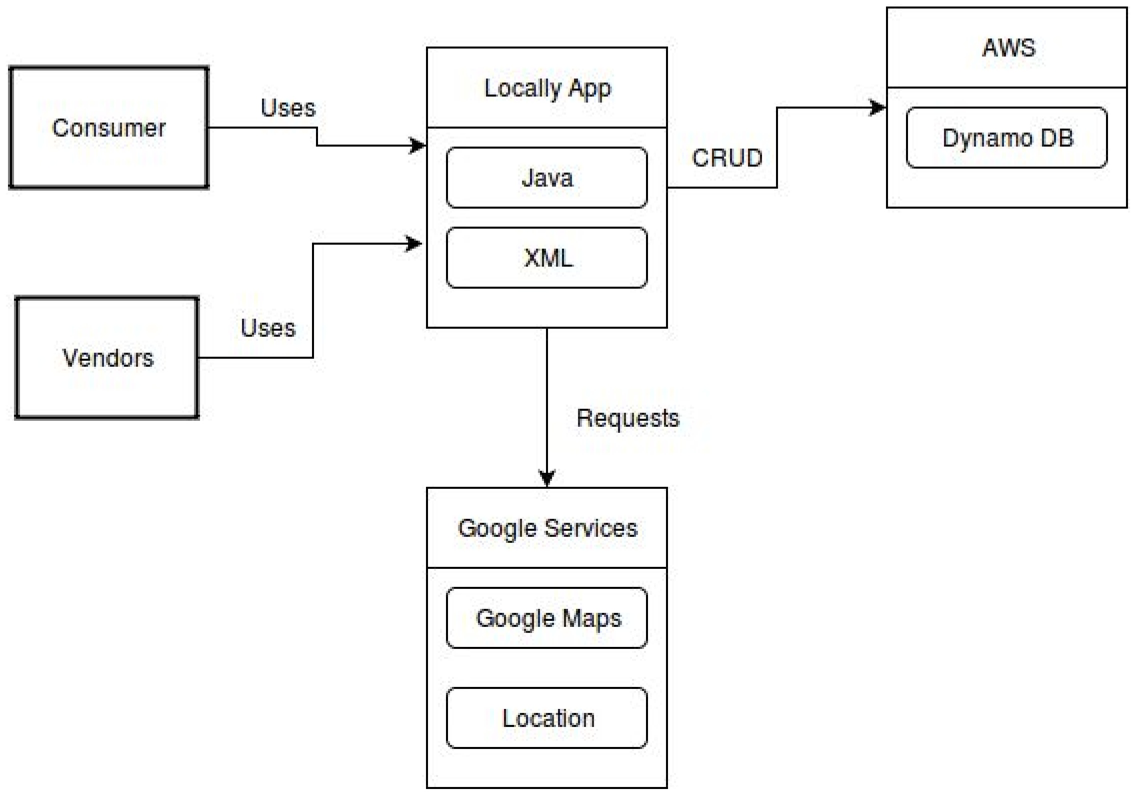
Model - The model is responsible for the business logic of the application which defines how data can be created, deleted, modified, and stored. On Android it is a data access layer, a AWS DynamoDB database in our case.

View - The view is a predominantly passive element that displays visual data, reacts to user actions, receives user input (events) and forwards it to the presenter. On Android, this could be an Activity, a Fragment, an android.view.View or a Dialog.

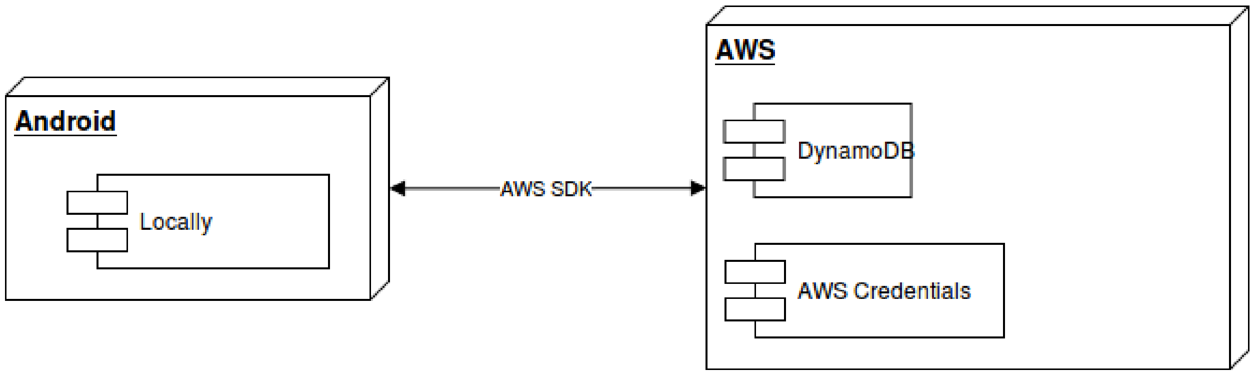
Presenter - The presenter serves as the middleman between the model and the view by receiving user events from the view, retrieving data from the model, and formatting data for display to the view. It is a codelayer dealing with all the background processing on Android.

The MVP model differs from the more traditional MVC model as the latter's View component can be more active and retrieve data directly from the Model. Thus, we chose to employ the MVP model for our project as it affords a greater separation of concerns between the three core architectural layers and allows for easier unit testing of the project.

Dynamic View



Deployment Diagram



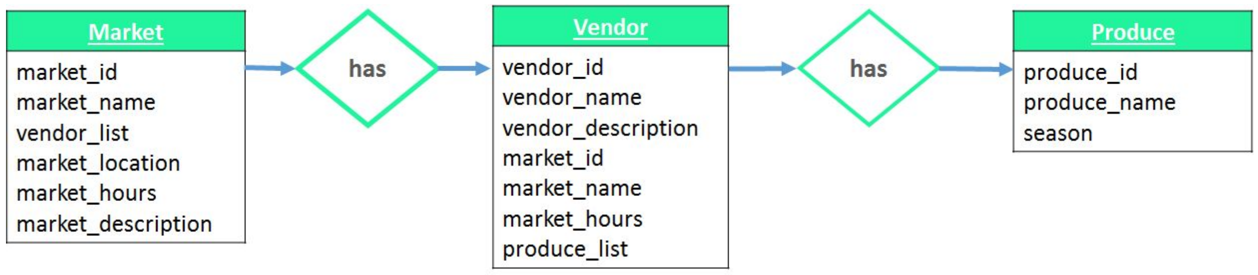
DATABASE DESIGN

DynamoDB

AWS offers an Android SDK for a DynamoDB allowing for CRUD (Creation, Reading, Updating and Deleting) database entries. By choosing DynamoDB we do not have to worry about load balancing, scaling, handling credentials, among other administrative tasks. Other potential solutions included Firebase or using REST but with support being excellent for DynamoDB along with the developers having more experience using AWS, DynamoDB was the natural decision.

Database Diagram

As of now the pseudo (NoSQL) Entity-Relationship diagram is as follows



GUI

As mentioned above, the user interface will be built using XML to design application views as per Google's Material Design guidelines. We will describe the design for main views below.

Home Page

We will place the navigation Sliding Sidebar (Hamburger) Menu with Navigation Drawer Icon in the top left corner of the home page. The menu shall contain main destinations for our app, such as map, settings, log in as vendor, etc. See the prototype on Image 1.

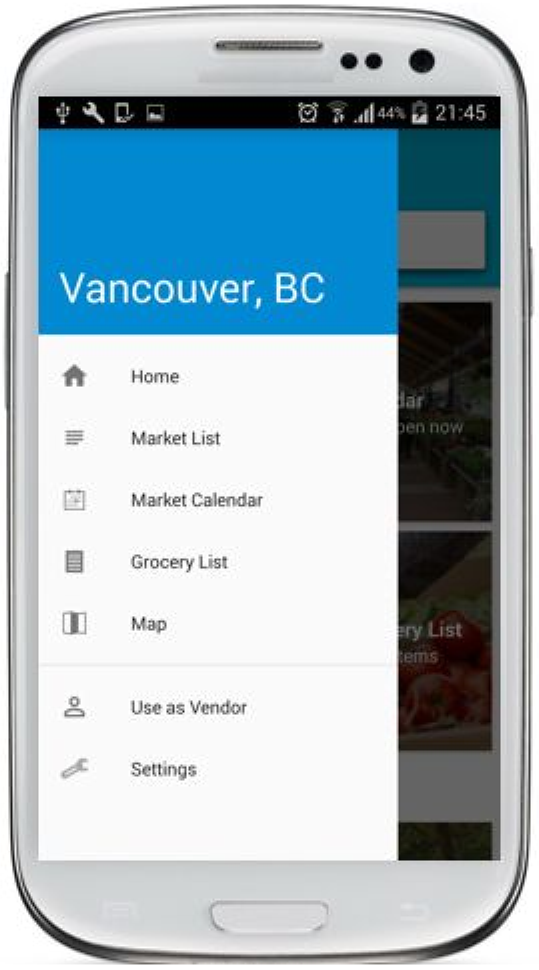


Image 1 - *Hamburger Menu*

Behind the navigation drawer, the home page will include several elements:

1. Search bar - for users to search a specific item and view its availability at vendors
2. Quick links section - links to the main features of the app consisting of markets list, calendar, a list of in season produce, and user’s grocery list; the thumbnail images are overlaid with text describing the link and relevant real-time data such as number of markets open or number of items in season

3. Markets Nearby section - if the user has allowed our app location permissions and has enabled their location services, they are able to instantly view the four closest markets to their area

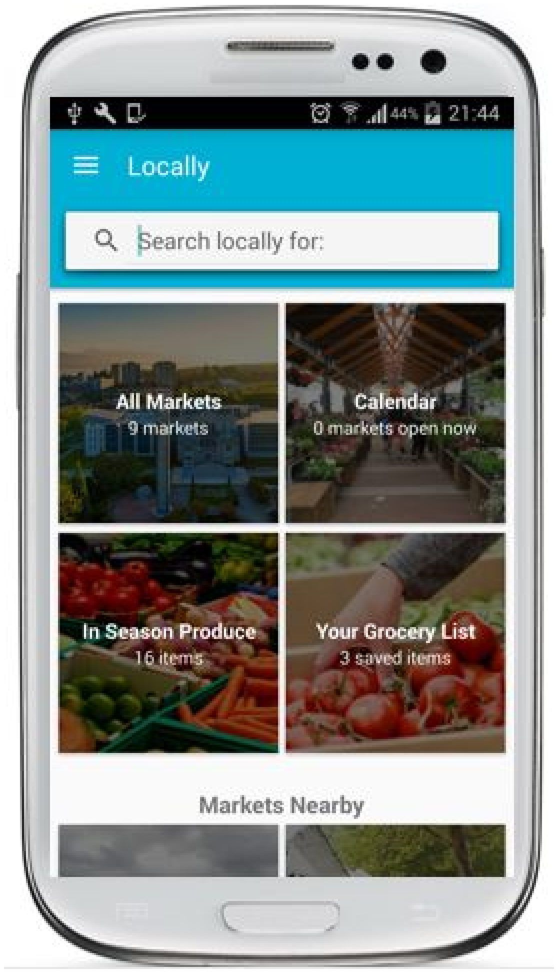
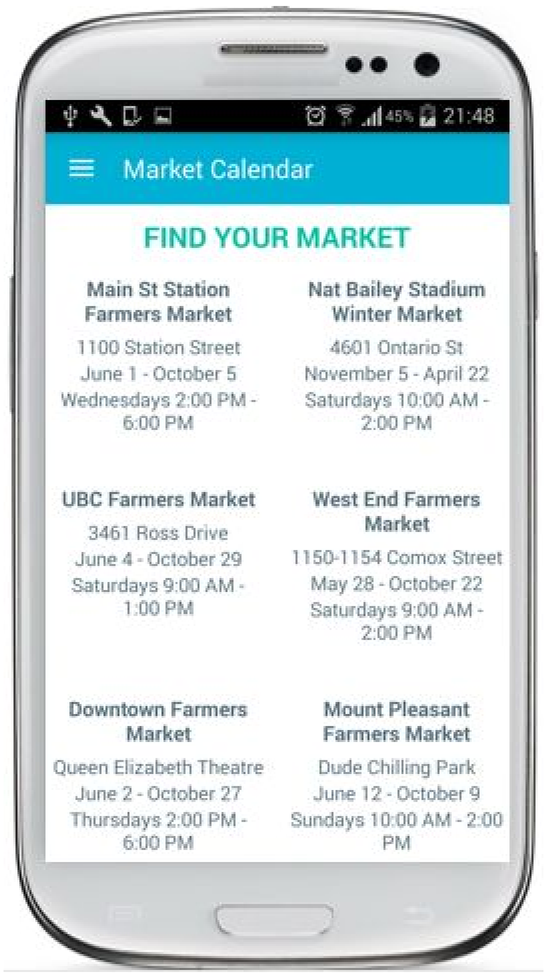


Image 2 - *Home Page*

Calendar View

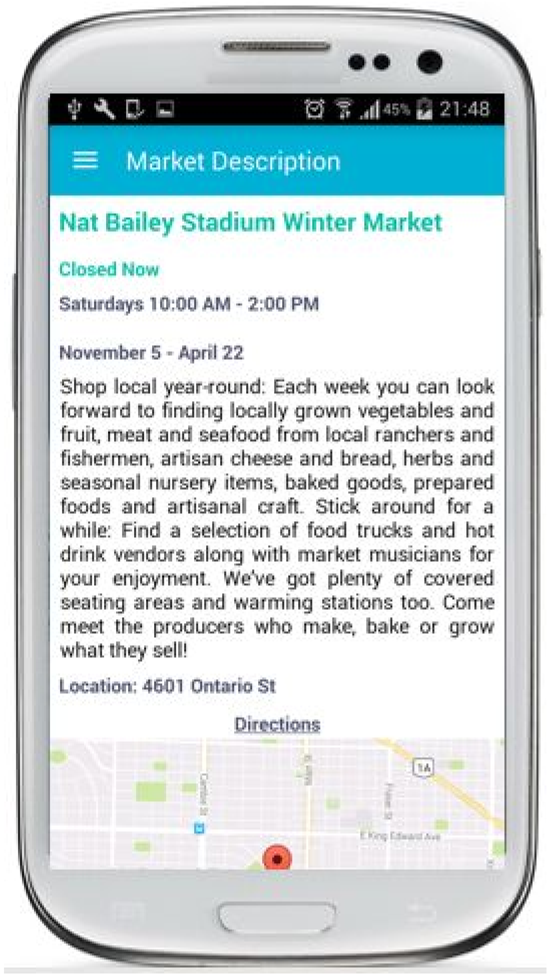
Calendar view button leads the user to the page displaying the list of market names, locations and hours/dates of operation. Once the user clicks on the market name they shall be directed to the page containing broader market description, directions link opening in separate Google maps window and the map view object displaying zoomed out location of the market. See the prototype design for the calendar view on image 3.



Market Calendar

FIND YOUR MARKET

Main St Station Farmers Market 1100 Station Street June 1 - October 5 Wednesdays 2:00 PM - 6:00 PM	**Nat Bailey Stadium Winter Market** 4601 Ontario St November 5 - April 22 Saturdays 10:00 AM - 2:00 PM
UBC Farmers Market 3461 Ross Drive June 4 - October 29 Saturdays 9:00 AM - 1:00 PM	**West End Farmers Market** 1150-1154 Comox Street May 28 - October 22 Saturdays 9:00 AM - 2:00 PM
Downtown Farmers Market Queen Elizabeth Theatre June 2 - October 27 Thursdays 2:00 PM - 6:00 PM	**Mount Pleasant Farmers Market** Dude Chilling Park June 12 - October 9 Sundays 10:00 AM - 2:00 PM



Market Description

Nat Bailey Stadium Winter Market

Closed Now

Saturdays 10:00 AM - 2:00 PM

November 5 - April 22

Shop local year-round: Each week you can look forward to finding locally grown vegetables and fruit, meat and seafood from local ranchers and fishermen, artisan cheese and bread, herbs and seasonal nursery items, baked goods, prepared foods and artisanal craft. Stick around for a while: Find a selection of food trucks and hot drink vendors along with market musicians for your enjoyment. We've got plenty of covered seating areas and warming stations too. Come meet the producers who make, bake or grow what they sell!

Location: 4601 Ontario St

Directions

Image 3 - *Prototype for Calendar View*

Map View

The map view presents a visual way to discover markets and vendors around a user. It utilizes the Google Maps API to display the locations of vendors as flags, which are linked to their respective vendor detail pages for users to learn more about a specific vendor.

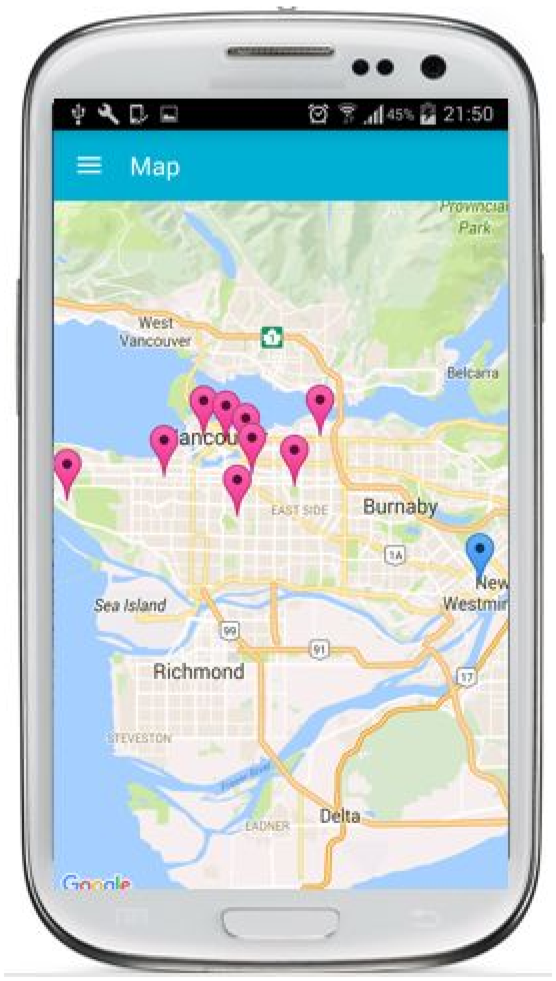
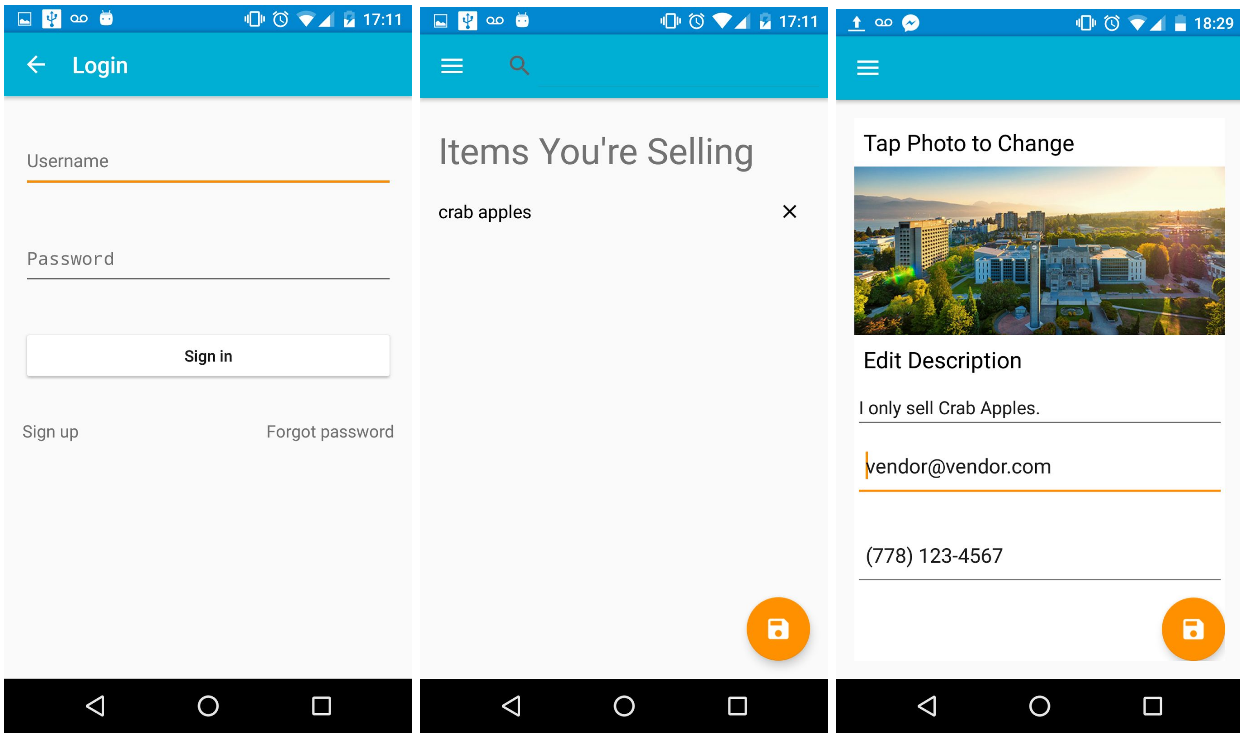


Image 4 - *Map View Screenshot*

Vendor Registration/Login

Once the user has selected the “Use as Vendor” button in the navigation drawer (hamburger), they will be brought to the vendor registration/login page. where they will be able to log in with their user credentials into our system to update their stock, or go through our registration process to register themselves as a vendor in our system.

From the user registration form we collect the username, vendor name, the market the vendor belongs to, their email and phone number. On successful completion of the fields, the user is added to our user pool on AWS Cognito. From here we can view their details and following verification of this vendor via us contacting them by email/phone we set their user to confirmed. Following this the vendor is able to login with their account. Once logged in the vendor can edit the items that they are selling, and their details.



crab apples

I only sell Crab Apples.

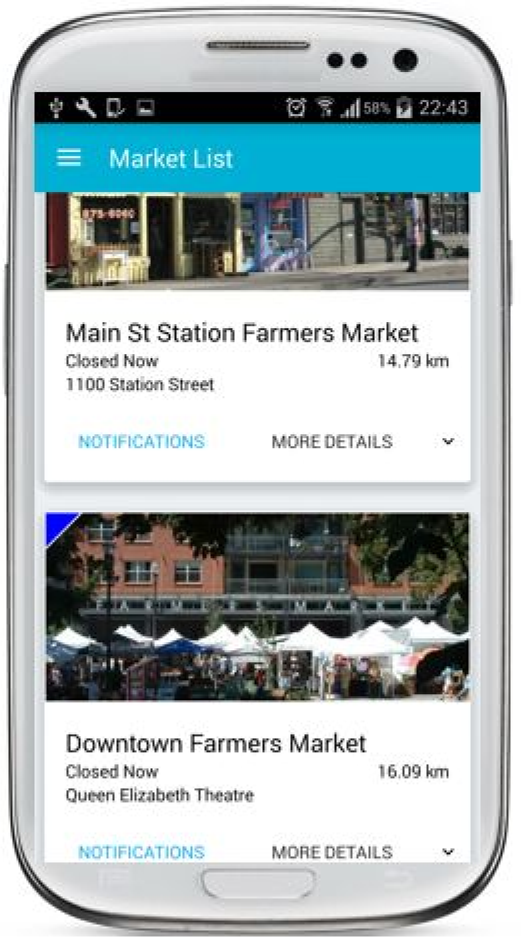
vendor@vendor.com

(778) 123-4567

Image 5 - *Vendor Registration / Login Screenshot*

Market List View

The market list view displays all available markets in Vancouver. Currently we have 9 markets in our database. If the user allowed our application access to location permissions, then the markets displayed will be sorted by the distance from the user with the market with the shortest distance at the top of the list. This view also displays the address of the market, the distance the market is from the user (if user allowed access to location permissions) and whether or not the market is currently open or closed. When the “More Details” button is clicked, the user gets access to more detailed information about the market, including the specific range of dates that the specified market is open and which hours it is open. As mentioned above, we designed the UI of our application according to Google Material Design guidelines, and thus, the Card View design of the market list follows Material Design guidelines.



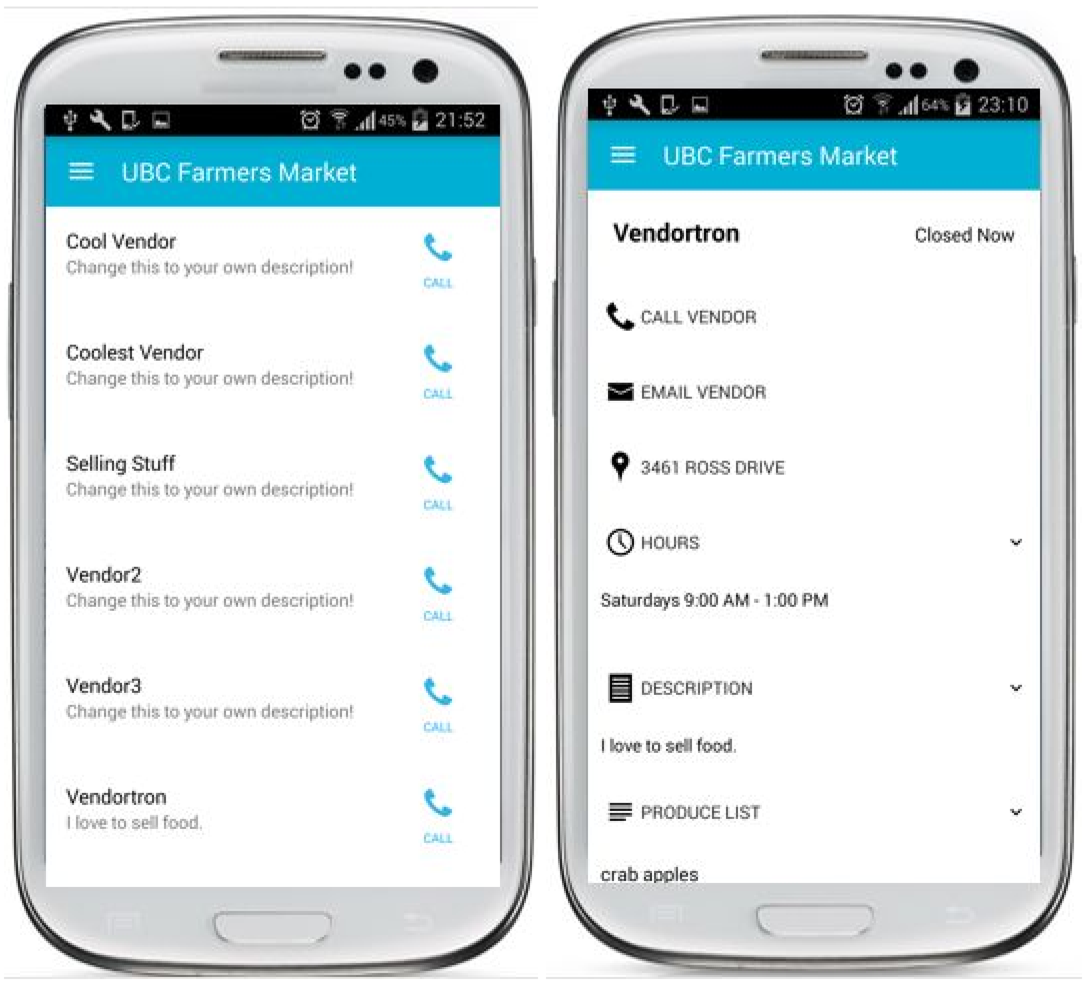
Main St Station Farmers Market
Closed Now 14.79 km
1100 Station Street

Downtown Farmers Market
Closed Now 16.09 km
Queen Elizabeth Theatre

Image 6 - *Market List View Screenshot*

Vendor List View

The vendor list view displays a list of vendors within a given market, with details such as vendor name and a short description of the vendor. This view also has quick access to a call function that will call the specified vendor at their provided telephone number if the "Call" button is clicked. If the user wishes to learn more about any vendor within the vendor list, they can double click a vendor list item to open up the vendor detail view. The vendor detail view is shown below in Image 7 alongside the vendor list view. Within the vendor detail view, users will be able to view all of the details mentioned above, as well as additional vendor information such as hours, and a produce list that describes the list of produce that they sell. The user will also have access to two methods in which they can contact this specific vendor, via phone or via email.



Cool Vendor
Change this to your own description!

Coolest Vendor
Change this to your own description!

Selling Stuff
Change this to your own description!

Vendor2
Change this to your own description!

Vendor3
Change this to your own description!

Vendortron
I love to sell food.

Vendortron Closed Now

3461 ROSS DRIVE

Saturdays 9:00 AM - 1:00 PM

I love to sell food.

crab apples

Image 7 - *Vendor List View and Vendor Detail View Screenshot*

These were the major views that we will use in our design. Other views, like list of search results or list of produce in season will be used in our application as well.

DEVELOPER-SPECIFIC INFORMATION

Source Code Access

In order to access the source code fork the repo from https://github.com/djung460/Locally. Sensitive files ie. API Keys and AWS Configurations are separate and are only available on request.

Design Patterns Used

As mentioned, we have chosen to employ the Model-View-Presenter (MVP) design pattern for our project. We have implemented this by organizing our Java classes into the appropriate Model, View, or Presenter packages.

Observer Design Pattern - Callback Mechanism

In this project the AsyncTasks use a callback mechanism to signal to the UI that the process is done. This allows the AsyncTask to simply pass codes and/or objects to the UI thread which then takes care of updating the UI based on these items. This encapsulates the background process and offloads it from the UI thread allowing for a smoother experience.

Code Structure

The project is divided into the following structure:

Java Code:

Package Names				
AWS	Model	View	Presenter	Utils
This package contains code that handles all things AWS with our app.	Each class here is a representation of what is stored in the DynamoDB.	These are all things related to what the end user sees. This includes Activities/Fragm ents, things related to UI elements ie. Custom Adapters	This is our interface with the database. Here we make Asynchronous calls to the database to fetch and handle the data that items in View can then use.	This package contains utility functions. For example decoding a string into Date objects for working with market hours. These classes have static functions that we can then easily test with our unit tests.

XML and Resources:

Package Names			
Drawable	Values	Layout	Menu
The Drawable package is used for	This package contains such things	This is where we keep all the layouts	Similar to the Layout package, this is

resources like icons and images.	as String values, API keys, Constants etc.	for our Activities, Fragments and user defined views	slightly more specific as it handles layouts related to menus. Ie. Navigation Drawer items.

DEVELOPING WITH AWS SDK

DynamoDB

The application currently uses 2 tables in our AWS DynamoDB.

1. Market Table
    - Partition Key: Market.Name
2. Vendor Table
    - Partition Key: Market.Name
    - Sort Key: Vendor.Name

If there is only a partition key, no two items can have the same partition key value, if there is both the partition and sort key then there may be multiple partition key values but they must have different sort key values. By using both keys in the Vendor Table we are then able to reflect the behvaiour of this application of fetching vendors based on what market they are in or have the ability to search for a specific vendor in a market. What this follows is the non relational database design principle, where our table's keys were designed based on the queries they would receive. Within the application these tables are reflected in our Models package. As an example our Vendor class is defined below with the primary partition key and the range key. The other non-key attributes would also be listed below in a similar manner.

```
@DynamoDBTable(tableName = "Vendor")
public class Vendor {
   // Name of the market it belongs to
   private String marketName;
   // Name of the vendor
   private String name;
   // ... Other attributes
   @DynamoDBHashKey(attributeName = "Vendor.MarketName")
   public String getMarketName() {
       return marketName;
   }
   @DynamoDBRangeKey(attributeName="Vendor.Name")
   public String getName() {
       return name;
   }
   // ... Rest of code
}
```

Simple Storage Service (S3)

We use this service to store market and vendor images. Since the Market and Vendor name are unique, we use that as a way of accessing the S3 Bucket with the images. For example, the names would first be removed of their whitespaces and if that results in market name “main_st” and the vendor name “fruit_stand”, then we would concatenate the two to get an s3 link like:
https://s3-us-west-2.amazonaws.com/locally-vendor-images/main_st-fruit_stand.jpg
With this link we use Picasso to painlessly load images from the web. Whenever the vendor needs to update their image they can then upload one from their device onto our S3 bucket and it will replace that unique link.

Cognito & IAM

These services are used for credentials. Cognito is where we store our vendor’s information. Currently we store their username, vendor name, the market they belong to, their phone number and their email. These are for verification and to fetch the appropriate information from the database upon login. Below is the Authorization flow of our application.

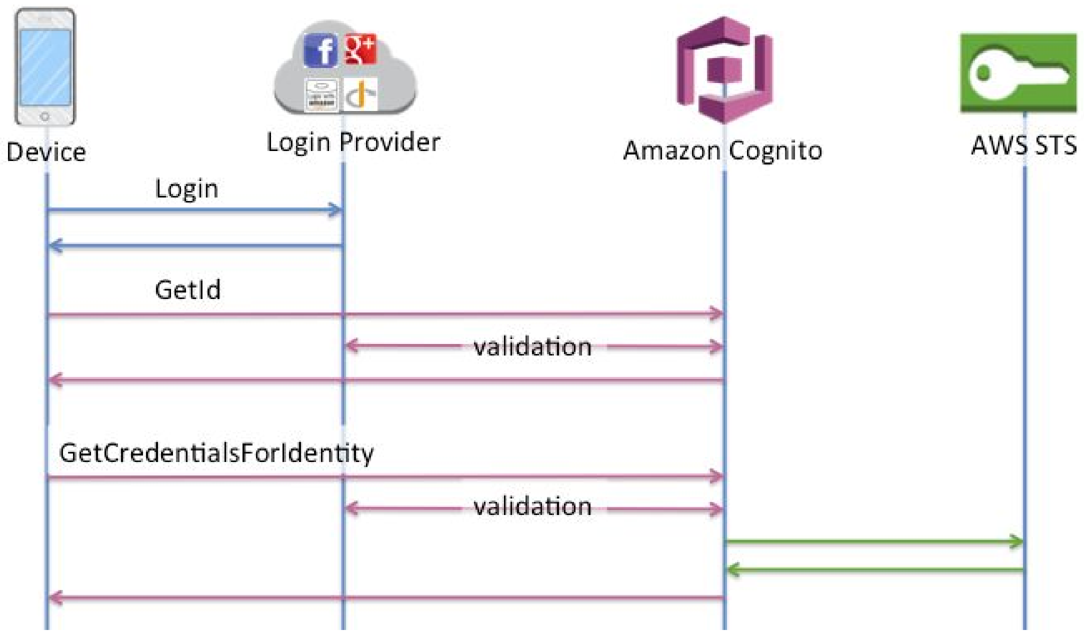


Image 8 - *Authorization Flow*

LOCAL DATABASES

Sqlite Local Database

The local database to store possible vendor items was used as opposed to a remote one due to the frequency of calls that will have to be made. While this may present the added problem of consistency this decision significantly improves item searching and allows us to make suggestions upon search.

Currently the search suggestion feature uses a local sqlite database that contains a set of potential vendor items taken from a PLU document. The user or vendor is able to then make searches and see suggestions appear as they type. This was implemented uses Multipurpose Internet Mail Extensions (MIME) to fetch possible suggestions.

Multithreading

Any work that is separate from the UI and takes an extended period of time or is prone to exceptions are done in another thread. These include any network calls, loading large resources, etc. This project uses Android's AsyncTasks and the Futures and Callables method. Ideally everything should be done using AsyncTasks, as the testing framework Espresso fully supports it. Making this transition is one of the plans for refactoring.

INSTALLATION AND SETUP

As previously mentioned, our project's source code can be found at https://github.com/djung460/Locally. Anyone who wishes to check out our project can do so by forking or cloning the directory onto their local machine and importing the project to the Android Studio IDE. One can also directly load the project from our Github repository by clicking File -> New -> Project from Version Control -> Github.

Building the Project

Since we are currently in a development phase and hosting the project on a public Github repository, certain classes/files are not included for the public in order to maintain security. In particular, to facilitate a connection to the AWS database, the AwsConfiguration.java class must be obtained from us and placed in the AWS package. Additionally, the file api_keys.xml must also be obtained and placed in the res/values directory.

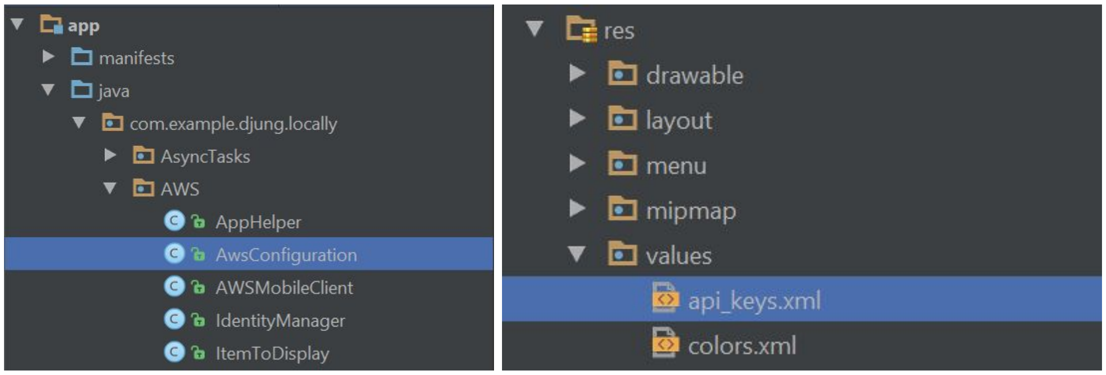


Image 9 - *Additional files needed and their target paths*

To build the app, select ‘app’ as the build configuration and click “Run” in the toolbar menu, i.e. app ▶, or go to Run -> Run ‘app’. Android Studio will then build and run the app with Gradle. The app will then be deployed to your choice of a deployment target which can either be a connected Android device or emulator. As the target SDK version of our app is for API level 23, we recommend running the app on an API 23 device/emulator (API 23 is also equivalent to Android 6.0 platform version or Marshmallow).

Testing the Project

The local unit tests created for our project involve testing the correctness of the app’s business logic including getting the number of markets currently open, list of closest markets, or checking if a market is currently open or closed. The local unit tests for our project are located in the Locally/src/test/java/ directory. To run the local unit test, we once again recommend using the Android Studio IDE. To run the tests, navigate to one of the unit test classes (e.g. MarketUtilsUnitTest). Run the tests by selecting the appropriate configuration and clicking “Run” in the toolbar, e.g. MarketUtilsUnitTest ▶, or selecting the “Run Test” icon in the sidebar next to the class declaration, e.g. `public class MarketUtilsUnitTest {`.

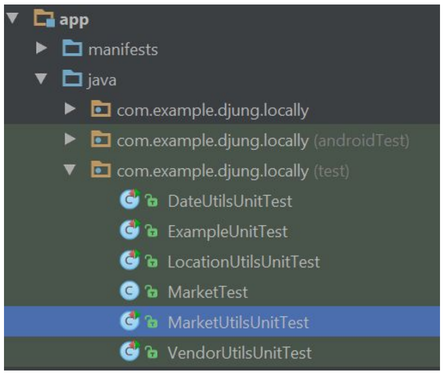


Image 10 - *Location of the local unit tests*

The instrumented unit test for our project are located in the Locally/src/androidTest/java/ directory. These tests must be run on an emulator or device.