Design Document for <<COOG >>

Group SB\_10

Omar Almehairbi: % contribution

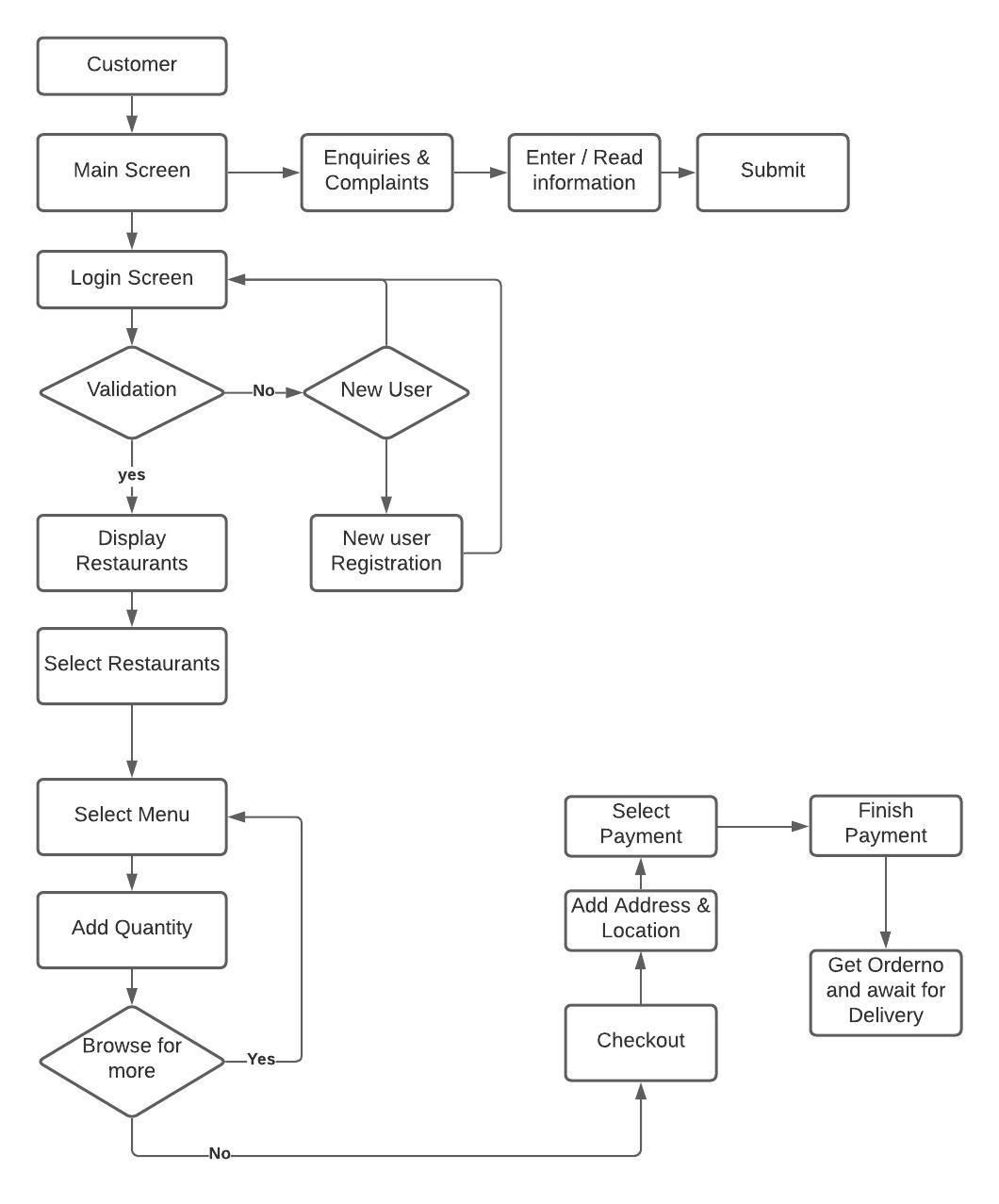
Omar Alsaedi:% contribution

Gerald Edeh% contribution

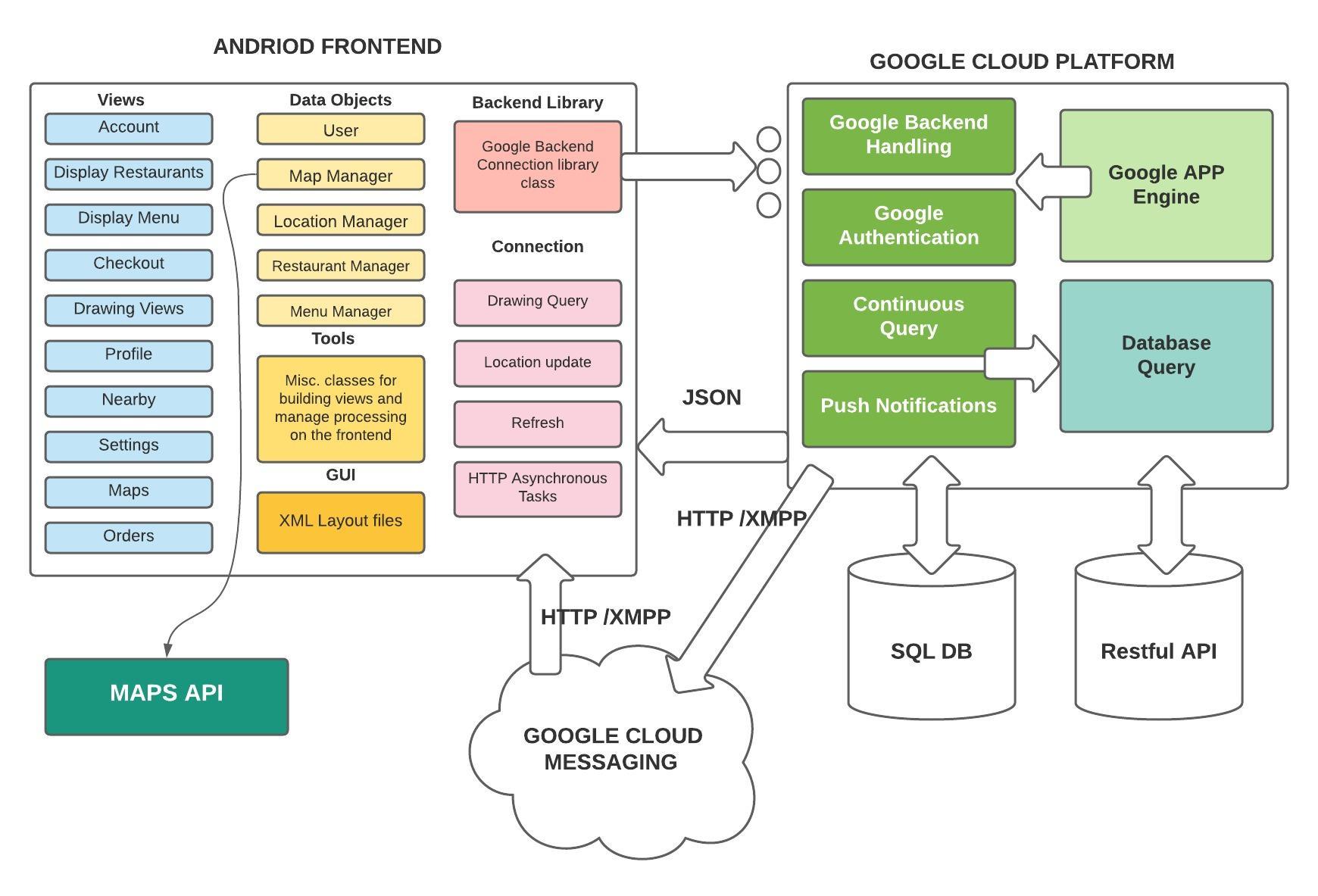
Chimzim Ogbondah% contribution

PUT THE BLOCK DIAGRAM PICTURE ON THIS PAGE! (Create the picture using lucid chart)

**Activity Diagram**



Use this third page to describe complex parts of your design.

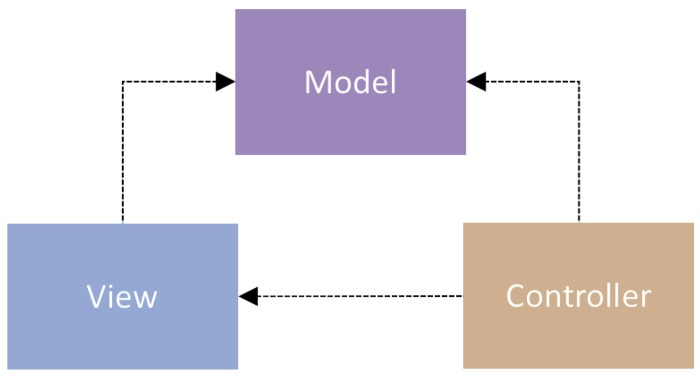
Design Description:

MVC model is applied to maximise testability;

**The Model-View-Controller Pattern**

In a world where the user interface logic tends to change more often than the business logic, the desktop and Web developers needed a way of separating user interface functionality. The MVC pattern was their solution.

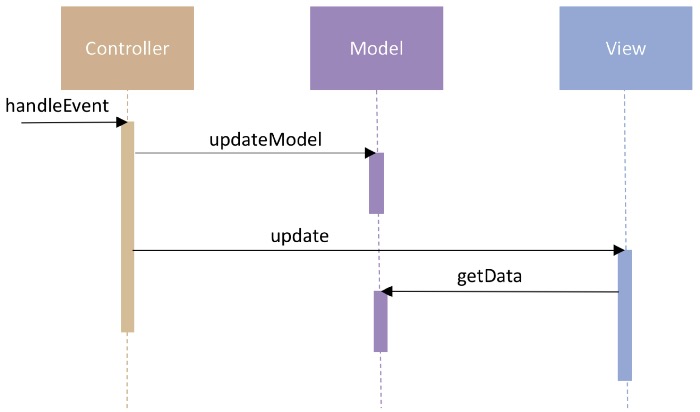
* **Model** — the data layer, responsible for managing the business logic and handling network or database API.
* **View** — the UI layer — a visualisation of the data from the Model.
* **Controller** — the logic layer, gets notified of the user’s behavior and updates the Model as needed.



Both the Controller and the View depend on the Model: the Controller to update the data, the View to get the data

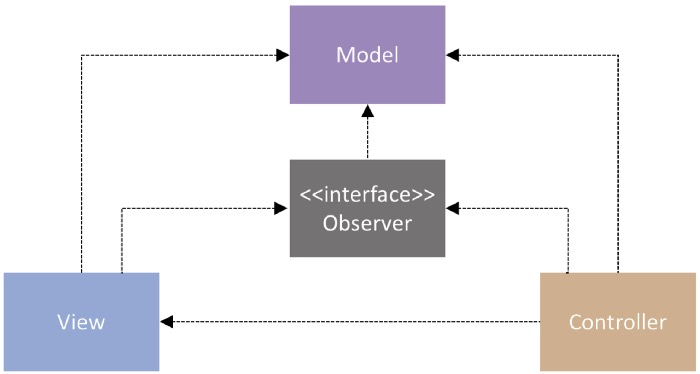
**Passive Model**

In the Passive Model version, the Controller is the only class that manipulates the Model. Based on the user’s actions, the Controller has to modify the Model. After the Model has been updated, the Controller will notify the View that it also needs to update. At that point, the View will request the data from the Model.

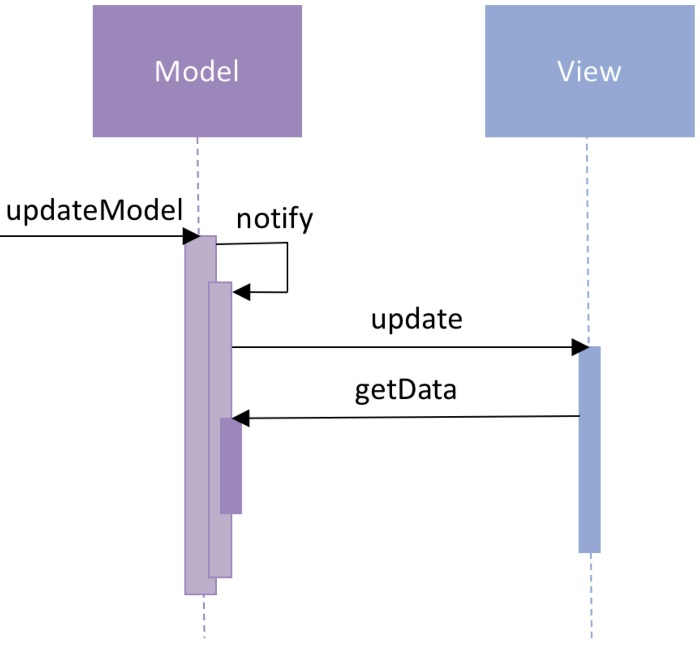


**Active Model**

For the cases when the Controller is not the only class that modifies the Model, the Model needs a way to notify the View, and other classes, about updates. This is achieved with the help of the Observer pattern. The Model contains a collection of observers that are interested in updates. The View implements the observer interface and registers as an observer to the Model.



Every time the Model updates, it will also iterate through the collection of observers and call the update method. The implementation of this method in the View will then trigger the request of the latest data from the Model.



Model  
Our app utilizes the Google Cloud Platform for both its server-side processing (with

Google App Engine) as well as its data storage (with the Cloud NoSQL schema-less Datastore and Cloud Storage). We chose Google Cloud Platform because of its relative ease of integration with Android as well as its scalability for future use. We use Cloud Endpoints as a RESTful service to ease the communication between our Android client and the API server. We are building off of the Mobile Backend Starter sample code for an Android client communication with a web application backend. This enables us to send push notifications (through Google Cloud Messaging) to notify users about new drawings posted close to them, as well as provide user authentication through existing Google accounts. For the communication with the backend, there will be separate thread which will run in background and will make use of Asynchronous Tasks that will not interrupt the main thread; thus, the UI will be always available to the user even when the background thread will be communicating over the network.

Controller  
A user’s location will be passed to the server periodically. The following steps will occur

to complete a request to the server for drawings nearby a user’s current location.

1. There are 3 cases in which a client will submit a new query for drawings from the server:
   * 1.1.  On application load (initial data population of the GUI)
   * 1.2.  After moving a set minimum distance from the last query point (this bit of   
     distance information is kept track of client-side)
   * 1.3.  After a new drawing is posted in the user’s nearby area (the client will be notified   
     via Google Cloud Messaging of new drawings in the area)
2. A request for drawings on the server consists of a user’s current location (latitude and   
   longitude) and a radius. Server-side logic will calculate the distance from drawings and only return drawing information (containing a cloud storage key for the image) from the Cloud NoSQL Datastore with that set radius.
3. An HTTP request to the RESTful Cloud Storage will retrieve the binary data of the image referenced by the cloud storage key retrieved in the previous step.

View  
Our application’s front-end consists solely of Android clients; thus, we utilize a

combination of XML and Java classes to control the communication and population of our application’s graphical user interface. With DrawNear being data driven, the application needs code written on the front-end to retrieve nearby images and populate views dynamically based on this information returned from the server. The Map view utilizes Google Maps API, which will allow users to view their current location as well as display markers indicating nearby drawings. Other views provide users with the ability to create and view posts, view their profile, and edit settings.