CPEN 321 - M4

Parsa Riahi
72762974

SnackBud

Main Modules of Components:

This section will be split into 4 categories to match the diagram of module dependencies in the third section. The front-end modules will be in the “Views” sections, with the client side back-end controllers and caches in the “Control” section, the Server side back-end in the “Services Layer”, and lastly the databases and 3rd party APIs will be under the “Data Layer” section. The single responsibility principle as all UI views are related to a single core idea in the app, and the rest of the modules all serve exactly one essential purpose each. The least knowledge principle is maintained by having all modules only talk to related modules who immediately call them or are called by them (direct neighbours only). One example is the separate caches, service layer services, and databases for {Meet-ups, Reports} vs for {Recommended Users, Recommended Restaurants}.

Views:	
Report View	This page allows the user to report any COVID symptoms they may have in order to notify those they came across through the app and aid in contact tracing. Push notifs will be sent to all affected parties and their entity icons on the map will be marked as high risk.
Map View	This page geomaps all recommended restaurants and recommended users to the current user in order to let them decide on where to create a meet-up. It is also the home page of the app, linking to all other views.
Restaurants View	This page displays all the restaurants, recommended ones included, and allows the user to get a better sense of their food options. The options here will be based on the user’s interest inputs. Meet-ups can be created from here too.
Profile View	This page lists the current user’s profile info, allows them to edit their profile, and lists the upcoming meet-ups which can be verified. When one clicks on another user on the map this page loads for that user allowing one to add them as a friend and create a meet-up with them.
Create MeetUp View	This is the main page that displays the meet-up creation process of selecting a restaurant, time, and guest. Once confirmed, push notifications will be sent to the guest.
Control:	
Request Manager	This is the main management unit between the UI and the backend services, handling all friending and meetup workflows, location requests, and it links to the caches. Thus this manager doesn’t have to worry about cache management.

Report & Meet Up cache	This cache links the two linked objects meet-ups and reports (of one user's meet-ups), which communicates with the Meet-up Service in order to manage meetups.
Recommended Object cache	This separate cache has knowledge only of the current recommended users and restaurants, as well as friends. It communicates with the Recommended Object Service in order to keep the recommended items up to date.
Push Notif Controller	This is the main mechanism which notifies users of new meetups, verified meetups, new friends or recommended users, recommended restaurants, and COVID reports.
Services Layer:	
User Account	This module is used to maintain all requests related to users, ie. friending a user, and editing one's own user. It links to Google Auth to maintain privacy.
Report and Meet Up Service	This service is used to manage meet-up creation and verification, links to the push notif controller to notify users of meet-ups and of COVID reports, updating the DB in the meantime. It also links to the Map and Calendar APIs to make meetup verification possible and to create events, respectively.
Recommended Object Service	This service is used to manage the recommendation algorithm of restaurants and users based on user interests. As such, it notifies users of its findings and requires access to the databases in order to get its input data. It needs access to Maps so that nearby elements can be displayed on the map screen
Report Message Bus	Bus to notify users of contact tracing reports in order to not overload the server with many notifications all at once, only when the users in the report open the app.
Data Layer:	
User Database	Database of all users, interests, lists of friends. Abstracted from meet-up DB.
Report and Meet Up DB	Database of all meet-ups and COVID reports. Abstracted from User Database.
Google Auth	3rd Party API used for user sign up and login, requiring a user to be signed in before viewing the main pages.
Google Maps	3rd Party API used to plot the restaurants and users onto the Map View, also to get the users' locations for recommendations and meet-up verification.
Google Calendar	3rd Party API used for the scheduling of meet-ups efficiently and abstracting away any notion of reminders and calendar managing.

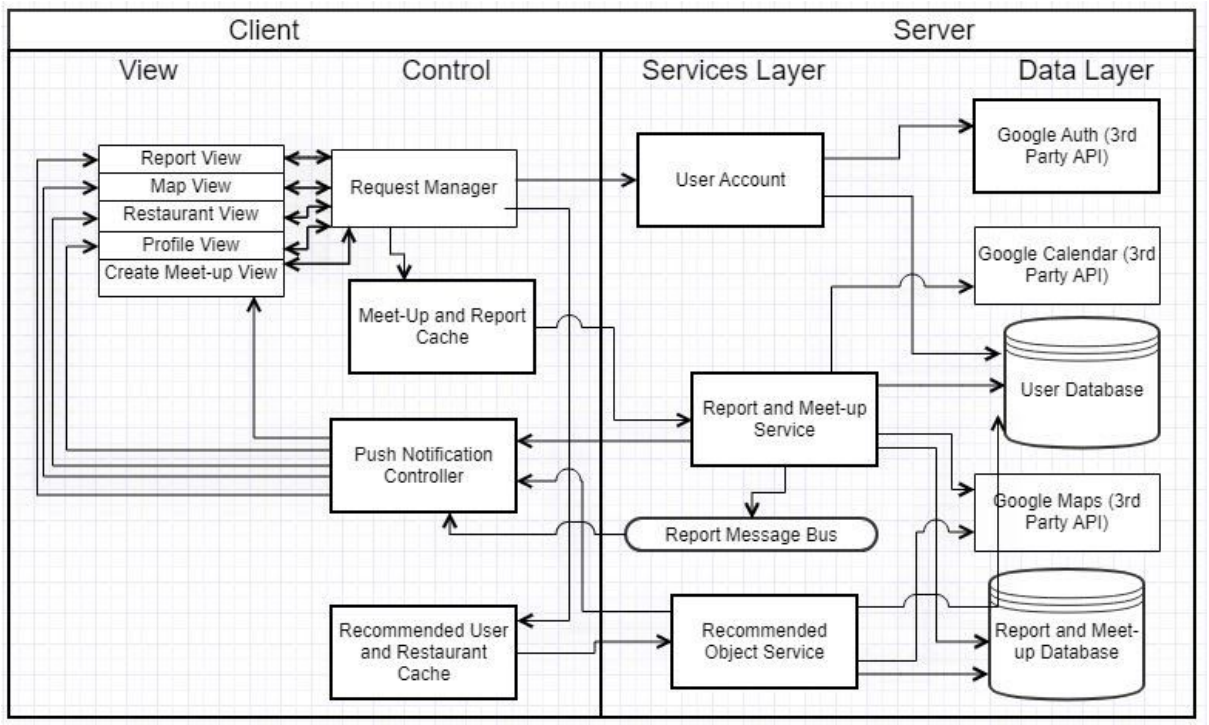
Interfaces of Each Component:

Views:

Report View	Report[] getMyContactTraces(), User[] getAtRiskUsers(), Restaurant[] getAtRiskRestaurants(), void postMyReport(bool showingSymptoms)
Map View	void loadMap(), User[] updatedUsers()
Restaurants View	Restaurant[] updatedRestaurants(), bool getCuisineRestaurants(Cuisine cuisine)
Profile View	void createProfile(String name,bio,phone,email,photoUrl, String[] interests) bool changeInterests(String[] interests), bool friendUser(User user), bool unfriendUser(User user), bool logout(), bool login()
Create MeetUp View	MeetUp createMeet(User host, Usr guest, Restaurant rest, DateTime timeOfMeet), bool verifyMeet(MeetUp meet, int verificationCode), bool cancelMeet(MeetUp meet)
Control:	
Request Manager	bool getAuth(), bool invalidateAuth(), void updateViews(int viewNumber), all other GET, PUT, POST, DELETE are run through here.
Report & Meet Up cache	Report[] getReportCache(), MeetUp[] getMeetUpCache()
Recommended Object cache	Restaurant[] getRecommendedRestaurants(bool filterByFav, Cuisine filterCuisine), User[] getRecommendedUsers(), User[] getFriends()
Push Notif Controller	void sendCreateMeetUpNotif(), void sendVerifyMeetUpNotif(), void sendRecommendedUserNotif(), void sendRecommendedRestaurantNotif(), void sendReportNotif()
Services Layer:	
User Account	
Report and Meet Up Service	MeetUp[] getMeetUps(), Report[] getReports, void deleteMeetUp(MeetUp meetUp), void deleteReport(Report report), void addMeetUp(MeetUp meetUp), void addReport(Report report), void updateMeetUp(MeetUp meetUp),
Recommended Object Service	User[] getUsers() User[] performMLClustering(User[] allUsers), Restaurant[] getRestaurants() Restaurant[] performAprioriAndGetAllCuisinesWithHighCorrelation(String[] userInterests, Restaurant[] allRestaurants)
Report Message Bus	bool pollForAtRiskUsers(Report report), void sendReportNotif(String userID, Report report)
Data Layer:	
User Database	DBRecord find(SQLquery), bool findAndUpdate(SQLquery, JSONString newVal), bool findAndDelete(SQLquery), bool createEntry(SQLquery, JSONString newVal)
Report and	DBRecord find(SQLquery), bool findAndUpdate(SQLquery, JSONString

Meet Up DB	newVal), bool findAndDelete(SQLquery), bool createEntry(SQLquery, JSONString newVal)
Google Auth	The functions relevant for user authentication from here, ie void onSignIn(googleUser) and void signOut()
Google Maps	The functions relevant to loading the map and retrieving user locations from here. i.e. LatLng getUserLocation(currUser), getDistanceBetween(LatLng currUser, LatLng userOrRestaurant), and MapView getEmbeddedMapView(LatLng center)
Google Calendar	The functions relevant to creating calendar events from here. (GET, PUT, POST, DELETE, PATCH)

Diagram of Dependencies:



Architectural Patterns for Full Stack:

The high level design pattern for this app is a compound one built off of the standard Client-Server Model. This is ideal as many clients are due to interact with a single server, and information like meet-ups, interests, and reports of one user can affect many others. The client side uses a Model-View-Controller to organize it into pages and Object models communicating with the back-end through the caches and request managers, and notifying users with the Push Notif Controller. This is especially useful in separate server requests from user interfaces from abstract classes, allowing for the maintenance of the single responsibility and least knowledge principles. The interfacing through the controller/manager also allows for a higher fan-in and lower fan-out on a module to module basis. The server side is designed to use RESTful APIs and services with any particular service having a layered architecture. This is very useful in interpreting the textual data of the databases and external

APIs in a stateless way, making things like the recommendation algorithm clearer. The notification structure for Reports in particular is more similar to a Message Bus due to the large number of devices which are due to receive up to data contact tracing data. This model makes the mosts sense here as opposed to the MVC model used for the oter notifications as it allows for devices to retrieve the data at their own pace (it is not pressing until the app is used) and thus this will also reduce the peak server load at any given time in notifying many users.

Language and frameworks used:

For the front end and app size backend of my application I will be using Java, with the server in MongoDB hosted on the Firebase Cloud provider. The choice behind choosing a native language for the app is two-fold. Firstly, I have already made robust cross-platform apps in Flutter, which is cross platform, so I want to get the understanding behind developing for a single platform, so that I can learn what optimizations each style has. I recognize that I may be sacrificing some of the hot reload capabilities possible on React Native but this is a small price for the easier API access, strict languages capabilities, and 3rd party libraries that one can get on a Native language. For the server, I chose MongoDB since it is the database I am most familiar with and it maps well to the OOP Java language as well as being very scalable compared to MySQL. Lastly, the choice for firebase comes from using it previously with my Flutter projects, having learned the features, UI, and app analytics/notifications of it previously.

Plans to Realize Non-Functional Requirements:

In order to create a meetup in 5 clicks the UI will be designed with one main map page which quickly links to all the app's features, from the main screen --> create meetup --> click twice for the restaurant and guest, and finally click on the confirm meetup creation button; total 5 clicks. This will add to app reusability and dependability. Next, to keep the map updating close to live for reliable data without the performance cost and low energy efficiency of continuous data pulling, I will pull the location data of the app user and their nearby recommended users/friends at a rate of once per 10 seconds. To allow for reliable contact tracing, which further adds to the safety and security of users on the app, their meet-up history will be stored for a liberal period of 21 days, long enough for anyone to notice any symptoms and to report it on the app. This data will only be shared if the user chooses to do so. Further security measures to protect users' privacy are to only show public user info of nearby recommended users and existing friends to any given user (and never their unique tokens or user IDs). Furthermore, the location data of users is only shown to the users who they are recommended to or are friends with while being within a 1km radius of each other, so that their location is only used for the planning and creation of meetups. Lastly, to add to the app scalability, I will list restaurants belonging to at least 20 distinct cuisines, which will attract personas who want to try new things, those who want to stick to their old favorites, and personas who are willing to pay for food at multiple different price points.

Algorithm for complex logic:

To get the recommended users, I will cluster them using the K-means++ Clustering algorithm based on their shared/similar interests along the following dimensions which are mapped from Strings to one-hot encoded numbers: Favourite Cuisines, Level Of Spontaneity in trying new foods, and Level of Comfort in meeting up in person. This will further be filtered by the user's proximity to the current User of the app after the algorithm has completed. This is shown in python which can be run as a script or converted into java later

```
# Intuition: how to identify clusters of features
# Choose the number of clusters, K
# Select any K random points (not necessarily from dataset) as the
centroids
# Assign each data point to the closest centroid (forms K clusters)
# while(reassignment of cluster data):
#     Compute and place the new centroid of each cluster
#     Reassign data points if possible
# We avoid the random initialization trap of choosing initial centroids
poorly with K-means++:
#     Choose one center uniformly at random among the data points.
#     For each data point x, compute D(x), the distance between x and the
nearest center that has already been chosen.
#     Choose one new data point at random as a new center, using a weighted
probability distribution where a point x is chosen with probability
proportional to D(x)2.
#     Repeat Steps 2 and 3 until k centers have been chosen.
#     Now that the initial centers have been chosen, proceed using standard
k-means clustering.
```

User[] recommendedUsers(User currUser):

```
# Importing the libraries
import matplotlib.pyplot as plt
import pandas as pd

# Importing the dataset
dataset = pd.read_csv('UserInterests.csv')
X = dataset.iloc[:, [3, 4]].values  # this is just to visualize clusters

# Using the elbow method to find the optimal number of clusters - only
needed initially to get the
# correct number of clusters
from sklearn.cluster import KMeans

wcss = []  # Within Cluster Sum of Squares, converges to 0 as numClusters
-> numPoints,
# so we choose the optimal number of Clusters at the pivot point of the
exponential relationship (elbow)
for i in range(1, 11):
    kmeans = KMeans(n_clusters=i, init="k-means++", random_state=42)
    # fit the data to the k-means++ model with a static seed for consistency
    kmeans.fit(X)
    wcss.append(kmeans.inertia_) # add the inertia of the last kmean fit
```

```
plt.plot(range(1, 11), wcss) # determine the elbow of the graph to get the
number of clusters
plt.title('The Elbow Method')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS')
plt.show()

# Training the K-Means model on the dataset
kmeans = KMeans(n_clusters=5, init='k-means++', random_state=42)
y_kmeans = kmeans.fit_predict(X)


# Visualising the clusters - only useful for debugging purposes (assumes 5
clusters, arbitrarily, as well as only 2 input features)
plt.scatter(X[y_kmeans == 0, 0], X[y_kmeans == 0, 1], s=100, c='red',
label="Cluster 1")
plt.scatter(X[y_kmeans == 1, 0], X[y_kmeans == 1, 1], s=100, c='blue',
label="Cluster 2")
plt.scatter(X[y_kmeans == 2, 0], X[y_kmeans == 2, 1], s=100, c='green',
label="Cluster 3")
plt.scatter(X[y_kmeans == 3, 0], X[y_kmeans == 3, 1], s=100, c='cyan',
label="Cluster 4")
plt.scatter(X[y_kmeans == 4, 0], X[y_kmeans == 4, 1], s=100, c='magenta',
label="Cluster 5")
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
s=300, c='yellow', label='Centroids')
plt.title('Clusters of customers')
plt.xlabel('Annual Income (k$)')
plt.ylabel('Spending Score (1-100)')
plt.legend()
plt.show()

# Then we take the cluster which contains the input user
return dataset.iloc[:, [0,0]].values.where(y_kmeans ==
y_kmeans.clusterOf(currUser);
```

Algorithm for complex logic 2:

The second complex logic case is the process of verifying a meetup, ensuring that both members of the meetup have attended the restaurant and that they have spent at least 10 minutes there so that the system is not cheated. If successful within an hour of the meet up start time, each user will be notified as such and their public profiles will be updated with their new number of meet-ups. When the user gets within the 200 meter proximity of the restaurant their contact tracing report is updated on the server, even if they haven't verified the meet-up. This makes verification useful for gamification of meet-ups as well as preventing malicious use of the app in creating too many unverified meets. Such a malicious user could be removed from the app. Thus the following will be the pseudocode algorithm once the meetup is created:

```
Void verifyMeetup(MeetUp meetUp, User currUser) {
```

```
    bool verify = false;
    onAppOpenListener() {
       if DateTime.now - 10 < meetup.timeOfMeetUp { break;} #too early
       if DateTime.now - 60 > meetup.timeOfMeetUp { break;} #too late
       LatLng currLocation = googleMaps.getDeviceLocation();
       Double euclideanDistance = LatLng.distanceBetween(currLocation,
                                                  meetUp.restaurant.latLng);

       while (true){
         if (euclideanDistance < 200.0 && currUser.userId == meetUp.hostId)
             { //host logic
           displayUniqueCodeOnApp(meetUp.verifyCode);
           updateContactTraceReport(currUser.uid, meetUp.guestId,
                                       meetUp.restId, meetUp.timeOfMeet);
           notifyMembers("Enter" meetUp.hostId + "\'s code",
                        meetUp.guestId);
           verify = checkDBForVerifyByGuest(meetUp.guestId,
                                         meetUp.SQLlocation);

       } else if (euclideanDistance < 200.0 && currUser.userId ==
                  meetUp.hostId) { //guest logic
           updateContactTraceReport(currUser.uid, meetUp.guestId,
                                       meetUp.restId, meetUp.timeOfMeet);
           onVerifyCodeEntered(String input) {
              if input == meetUp.verifyCode {
                 verify = true;
                 updateDBMeetUpVerified(meetUp.guestId, meetUp.SQLlocation);
                 notifyMembers(meetUp.guestId + "verified your meet-up",
                               meetUp.hostId);
              }
           }
       }
       if verify {
           updateCurrUserNumMeetups(currUser.uid, currUser.numMeets++);
           displayMeetUpVerificationMessage();
           break; # meetup made, stop polling/listening, exit function
       }
     }
}
```