Mobile Computing Report

Food Order Android Application

Stănășel Anca-Maria

Github Repository at: <https://github.com/annnca/FoodDelivery>

# About the project

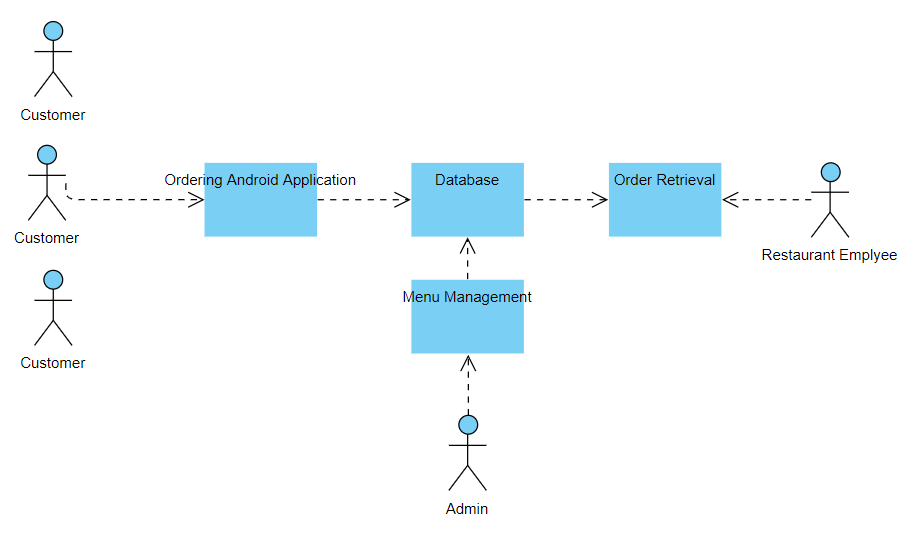


Fig. 1.1 System Model

The structure of the system can be divided into 2 main logical components:

* Mobile Ordering System- provides the functionality for customers to place their order and supply necessary details.
* Menu Management - allows the restaurant to control what can be ordered by the customers

**Product Function:**

The Online Food Order System application has the following basic functions:

**Mobile Ordering System Module**

This module provides the functionality for customers to place their order and supply necessary details. Users of the system, namely restaurant customers, must be provided the following functionality:

* + - Create an account.
    - Log in to the system.
    - Navigate the restaurant’s menu.
    - Select an item from the menu.
    - Add an item to their current order.
    - Review their current order.
    - Place an order.
    - Check the amount to be paid at delivery
    - Select the destination of the order
    - Enter promotion contests

**Menu Management System Module**

This module provides functionality for the power user-Administrator only. It will not be available to any other users of the system like Restaurant Employees or Customers. Using a graphical interface, it will allow an Admin to manage the menu that is displayed to users of the mobile ordering system:

* Add /update/delete food item to/from the menu.
* Update price for a given food item.
* View all disponible items
* Update additional information for a given food item.

Before customers can actually use this system, functionality provided by this component will have to be configured first. Once the initial configuration is done, this will be the least likely used component as menu updates are mostly seasonal and do not occur frequently.

# 2. Application wireframe

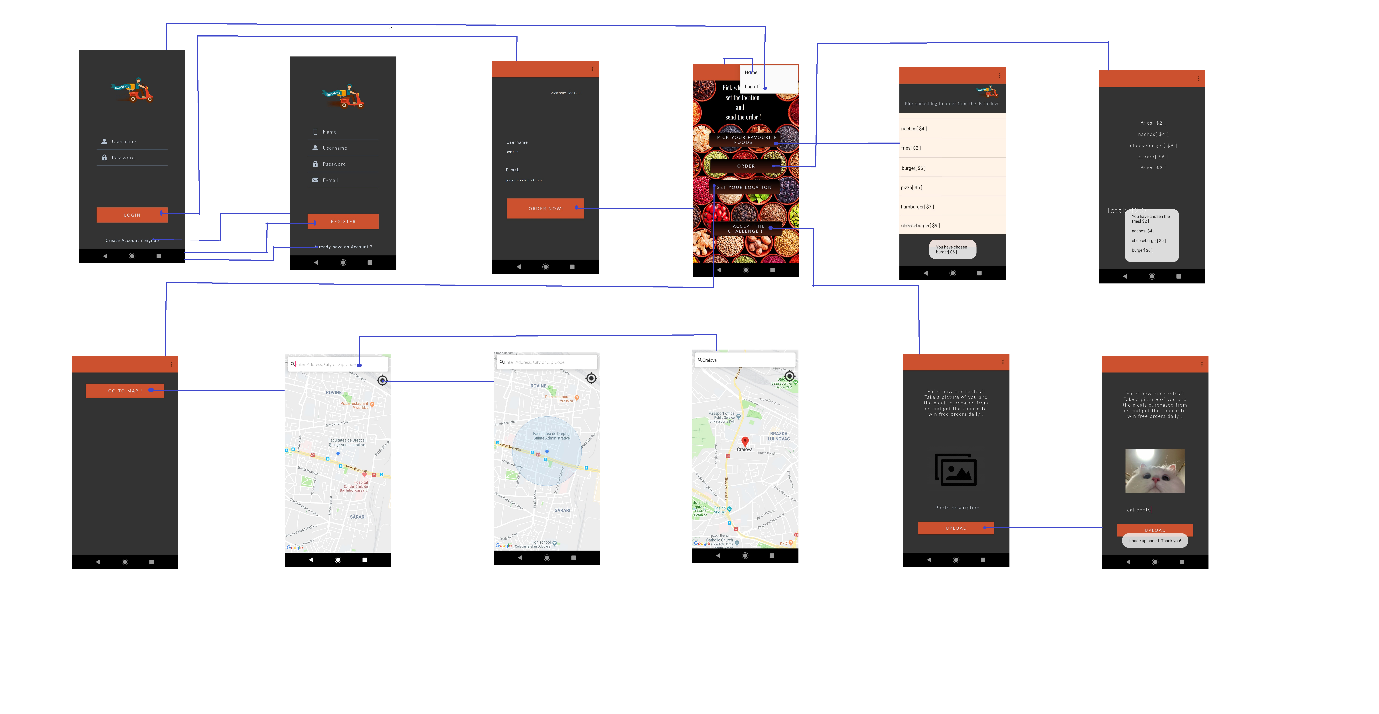


Fig. 2.1 Application Wireframe

The **LogIn** View can be switched to:

* + The Register View when Create Account? Register option is selected
  + The User Area View when Login button is clicked

The **Register** View can be switched to:

* + The Login View after all the fields have been completed and Register button is clicked
  + The Login View when Already have an account? option is selected

The **User Area** View can be switched to:

* Main View when Order now! Button is clicked

The **Main** View can be switched to:

* Foods View when Pick your favourite foods! button is clicked
* Order View when Order button is clicked
* Main Map View when Set your location button is clicked
* Image View when Accept the challenge button is clicked

The **Main Map** View can be switched to:

* Map View when Go to map! Option is selected

The **Map** View can be switched to:

* Own location View when the ‘locate’ button is clicked
* Searched location View when the search bar is used

The **Image** View can be switched to:

* Uploaded Image View when the image is chosen and the Upload button is clicked

All of the views, apart from Login and Register, have the upper toolbar incorporated with two possible view switches: Log Off to go back to the Login View and Home to go to the Main View.

The admin side view :

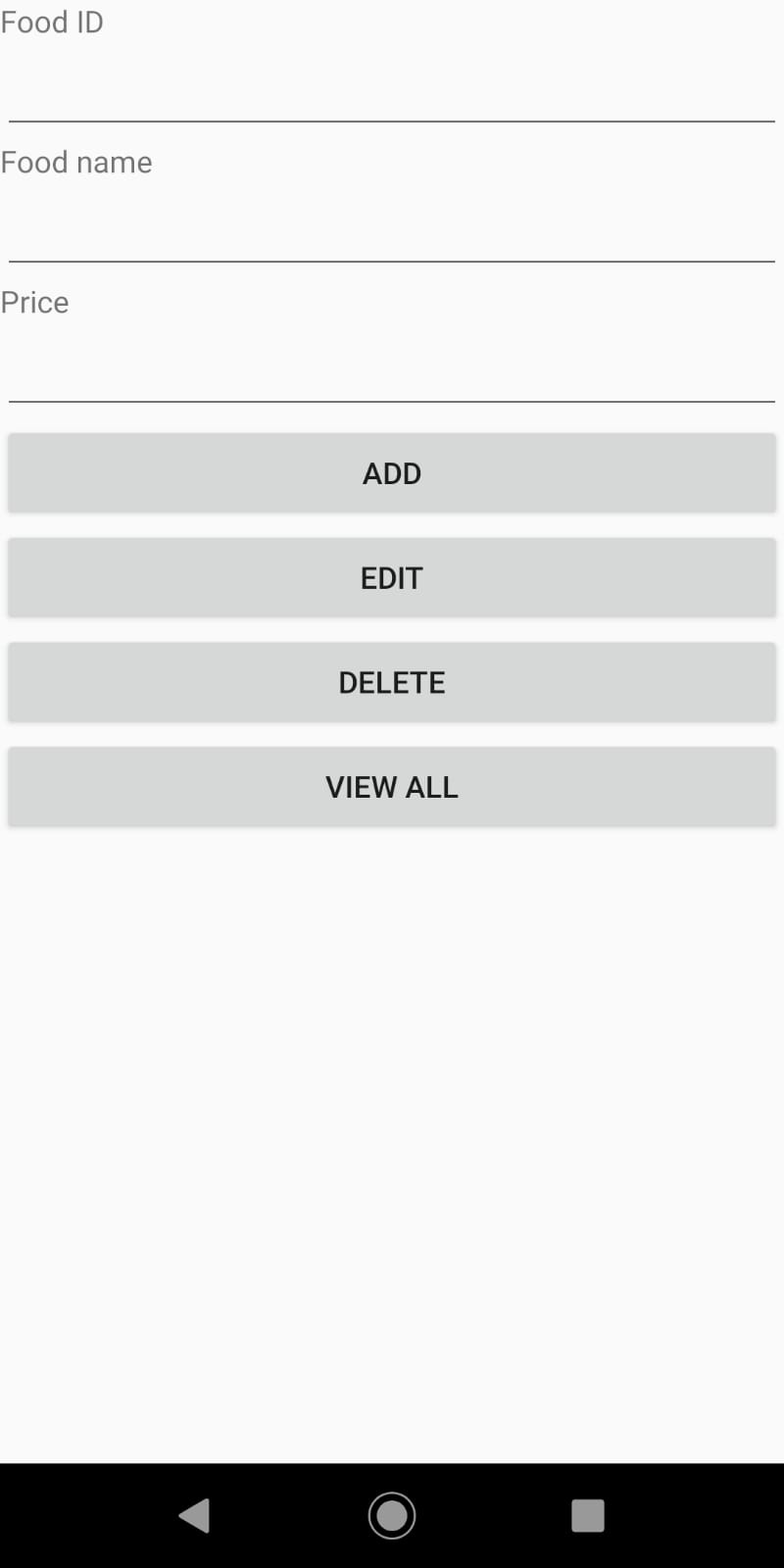


Fig.2.2 Admin View

# 3. Server Side Functionality

In order to create an online server to host the needed operations, the website for free hosting was used: <https://www.000webhost.com/>.

A database was created in order to store the users information:

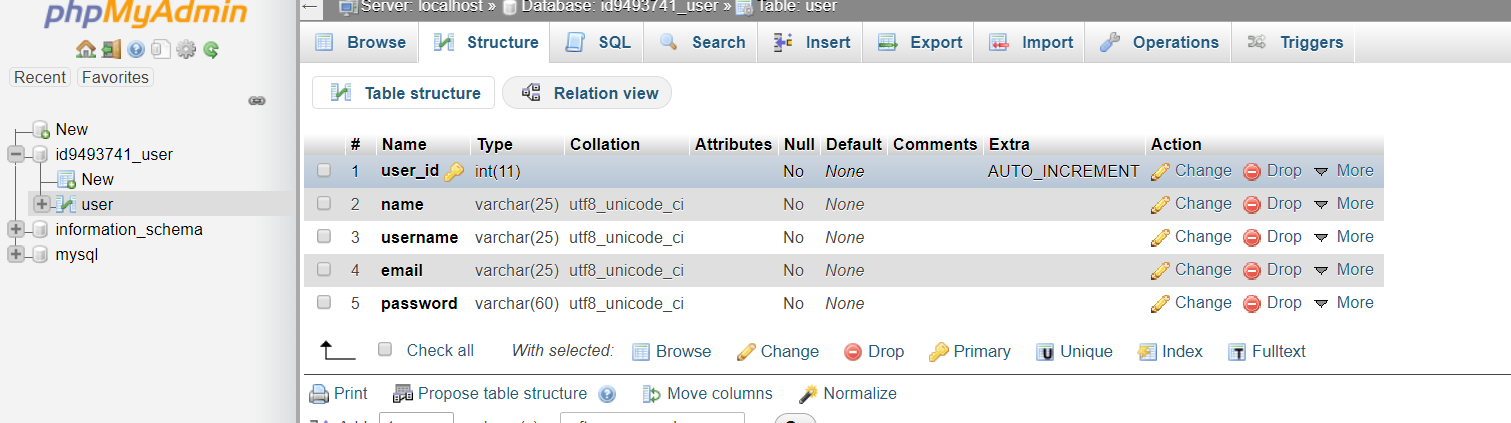


Fig.3.1 Users database

The server side functionality is split into:

Login functionality

Register functionality

Login Functionality:

The php script -***Login***.php- gets the username and password from the database and retrieves the corresponding user in order to check if the inputted password is the correct one. The result is stored and passed on through the ‘success’ variable.

**LoginRequest** class is used to hold the information of the server request that will be triggered when a login process is developing. It stores the LOGIN\_REQUEST\_URL which is mapped to the address where the .php script – Login.php- is found in the server’s files and the parameters of the request – username and password coupled inside a HashMap structure.

The response is sent back as a String. What the actual implementation inside the **LoginActivity** consists of: When the Login button is pressed: a request to the server is triggered, the php script is run. A ResponseListener is created in order to be received from the server. When the response is received(implemented in *onResponse*() overriden method), the result of the operation stored in ‘success’ variable is saved as a JSONObject .If the response was successful, then the UserAreaActivity is created and it gets the user information(name, email, username) from the JSONObject response.

Else, an AlertDialog is opened.

Moreover, *Volley* library is used in order to create a RequestQueue and add the Login Requests for dispatching the parsed response back to the main thread for delivery.

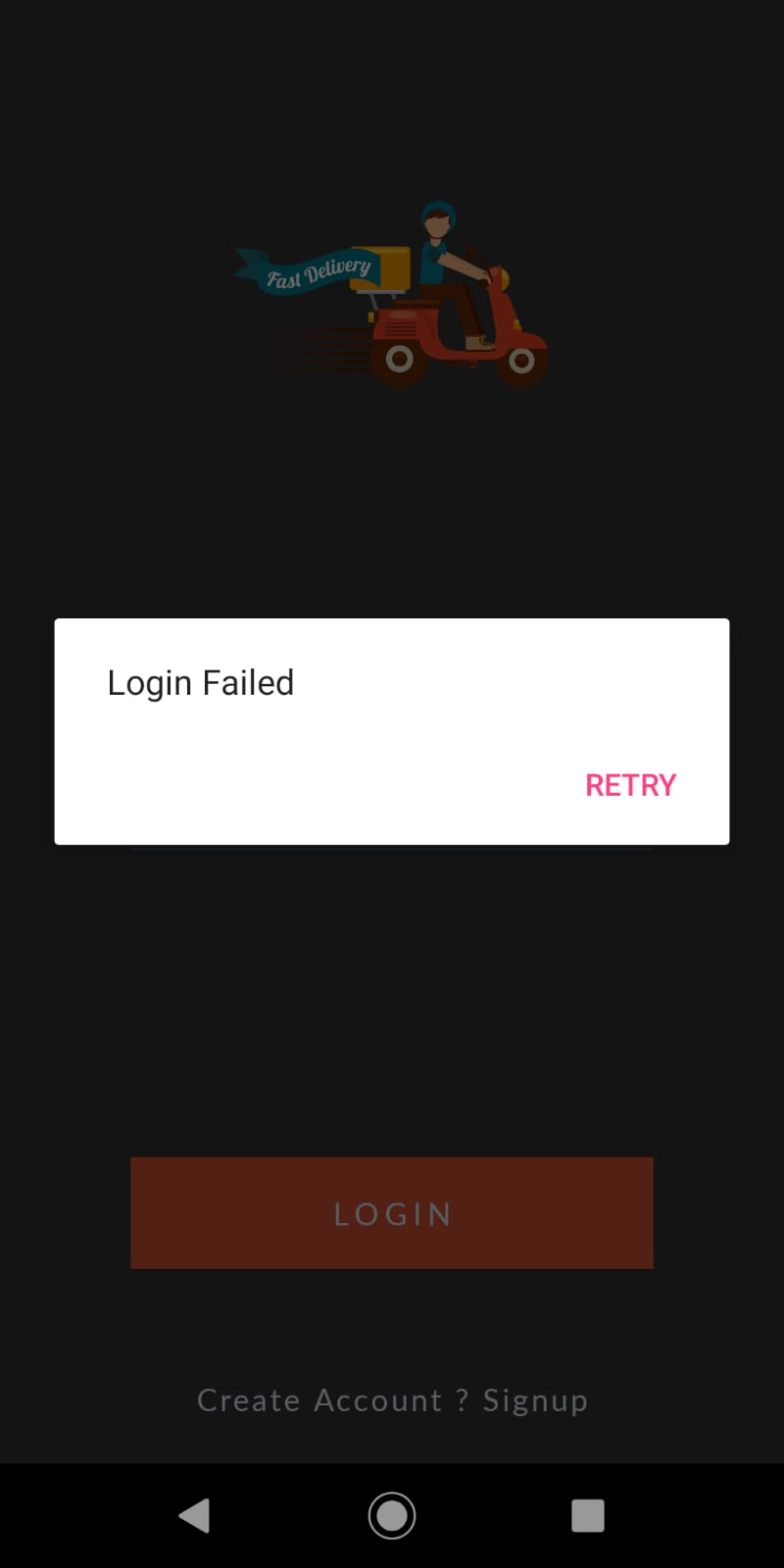
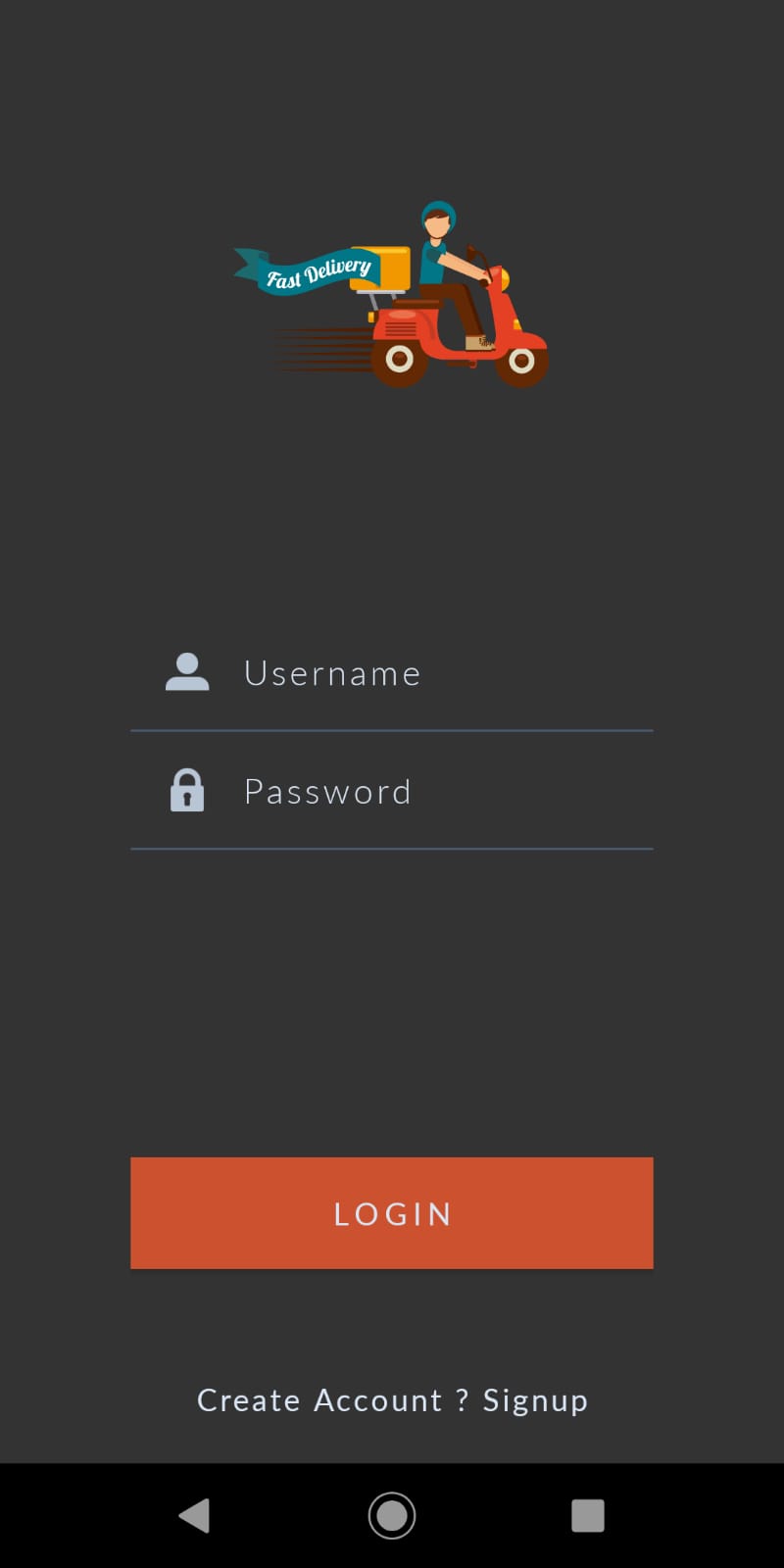
 

Fig.3.2 Error Message Fig.3.3 Login view

Register Functionality:

The php script – **Register**.php- is run whenever a register request is triggered to the server.

It first checks the username availability by going through all the users already existing in the data base (*usernameAvailable*() function) and then creates a password hash in order to encrypt the password. The used algorithm is imported from the **password**.php library which is to be found at : <https://github.com/ircmaxell/password_compat/blob/master/lib/password.php>.

If the request was successfully triggered, then the ‘success’ variable is set to ‘true’.

**RegisterRequest** class is used to hold the information of the server request that will be triggered when a register process is developing. It stores the *REGISTER\_REQUEST\_URL* which is mapped to the address when the .php script – Register.php- is found in the server’s files and the parameters of the request – username,name,email and password couple inside a HashMap structure. The response is sent back as a String.

Inside **RegisterActivity**: When the Register button is pressed: a request to the server is triggered, the php script is run. A ResponseListener is created in order to be received from the server. When the response is received (implemented in *onResponse*() overriden method), the result of the operation stored in ‘success’ variable is saved as a JSONObject .If the response was successful, then the LoginActivity is created.

Else, an AlertDialog is opened.

Same as in LoginActivity, *Volley* library is used in order to create a RequestQueue and add the Register Request for dispatching the parsed response back to the main thread for delivery.

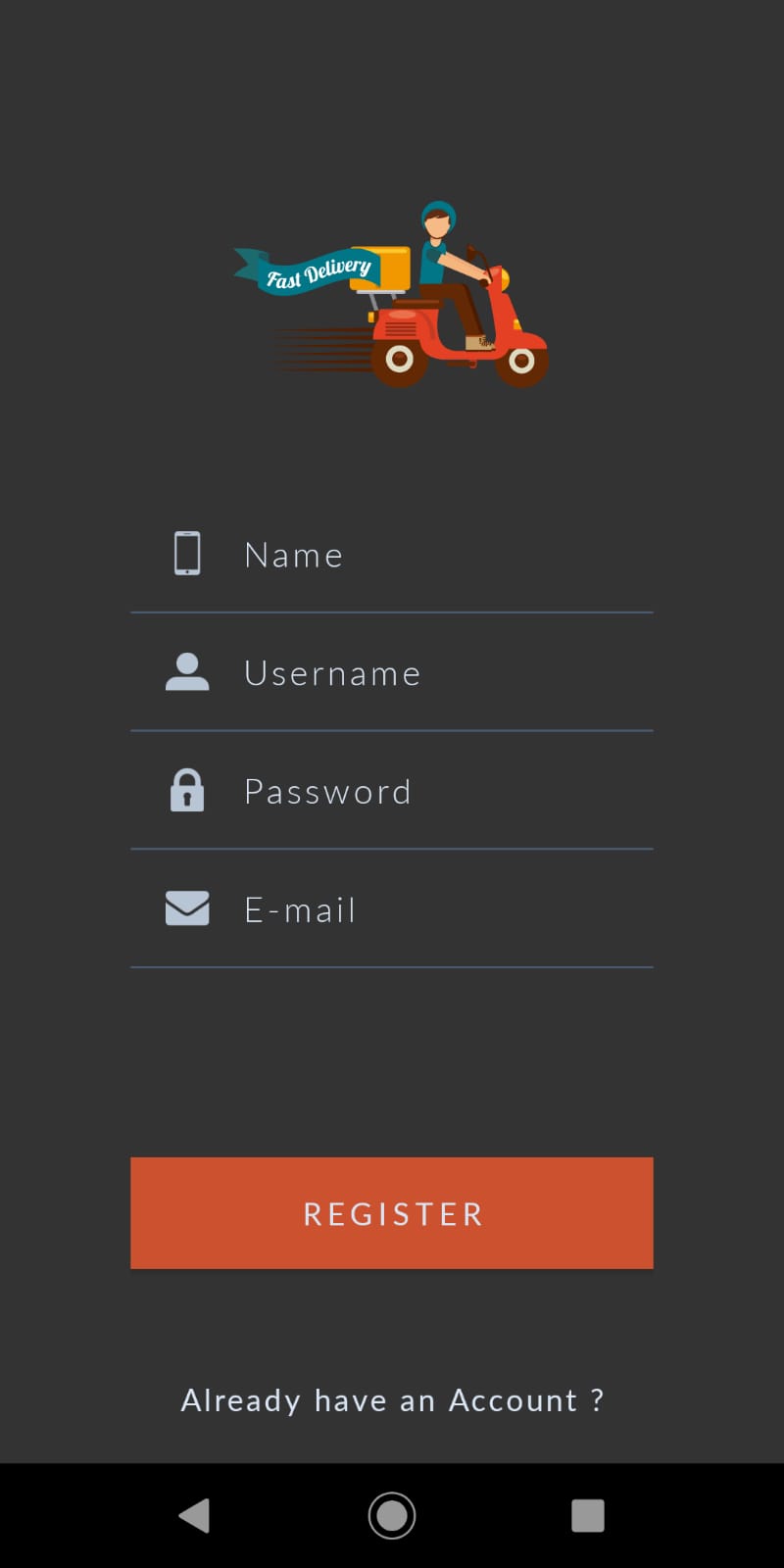
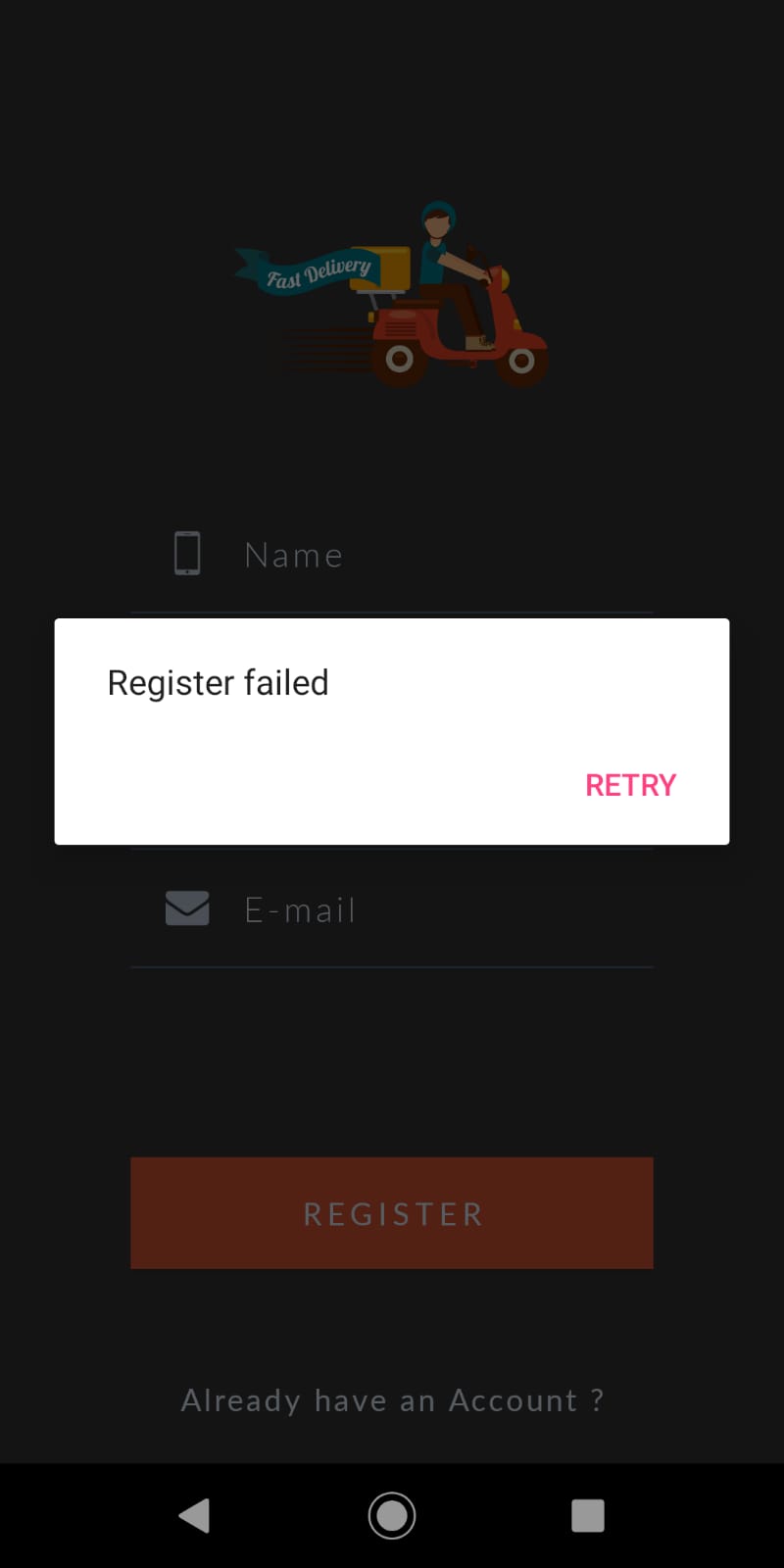
 

Fig.3.4 Register view Fig. 3.5 Register Error

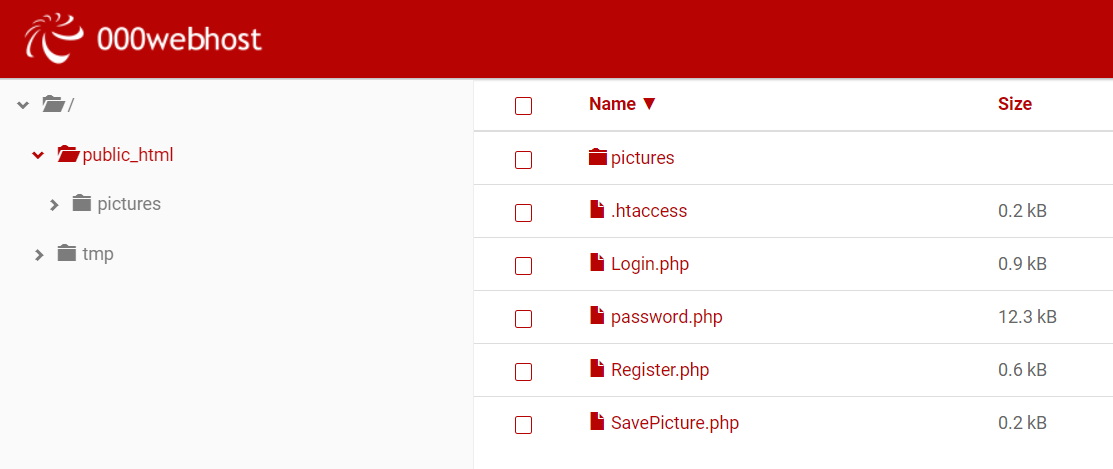


Fig. 3.6 Server files

# 4. Movement

NA

# 5. Images

The user is able to upload images from the phone’s gallery on the server.

This is done in two steps: first the photo is chosen from the device storage and the, the actual upload is done. Both are implemented inside **ImageActivity**(implements View.onClickListener).

When the image to upload is selected, it is needed to start an activity and get as result the actual photo. Of course, startActivityForResult() is used with parameters set as : intent – an Intent with the action to be performed set as ACTION\_PICK and the data to operate on as a Uri set as MediaStore.Images.Media.EXTERNAL\_CONTENT\_URI;requestCode – RESULT\_LOAD\_IMAGE. The user will, then, open the galley or any other preferred storage place and choose the photo which will appear in the application.

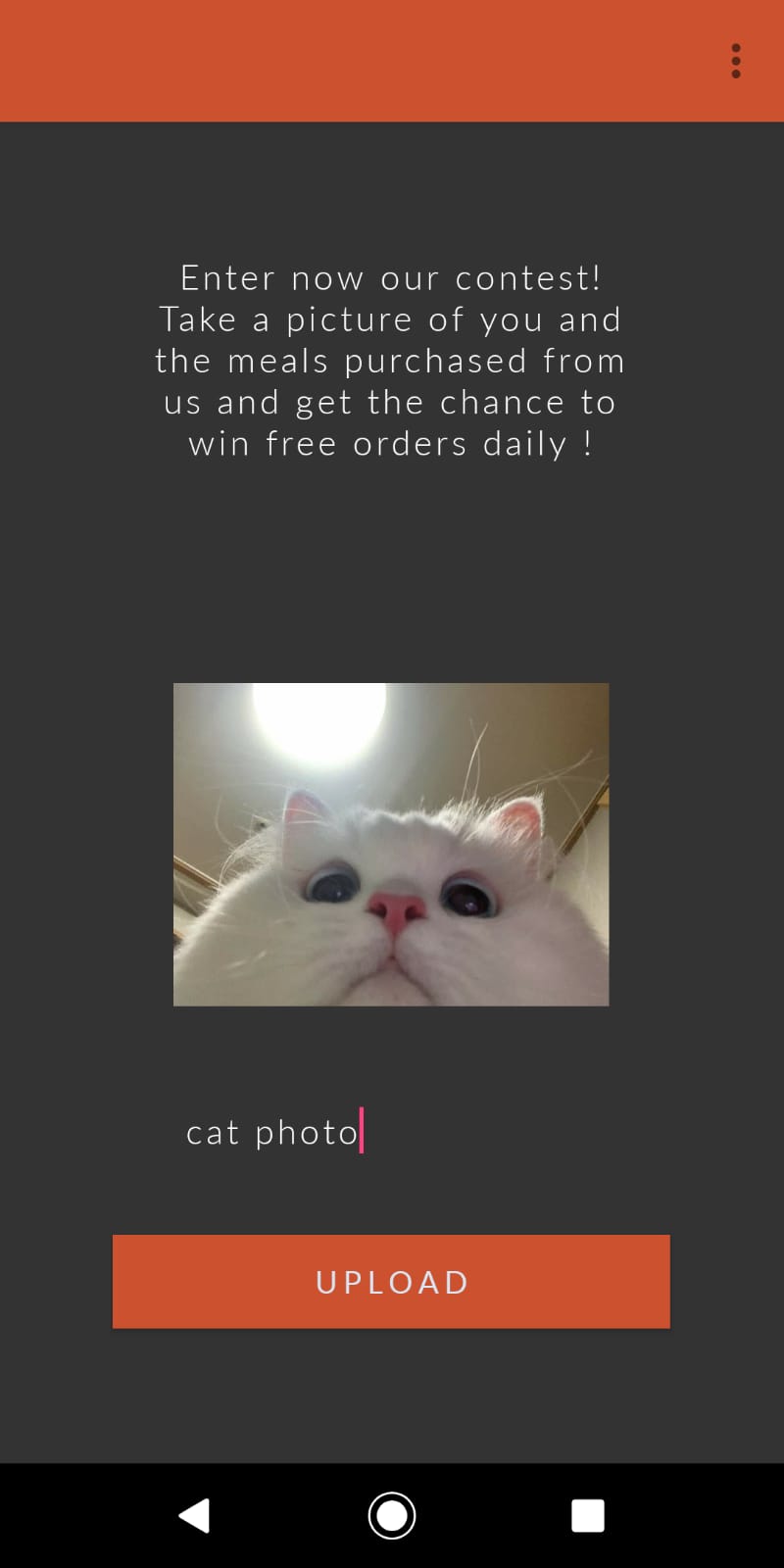
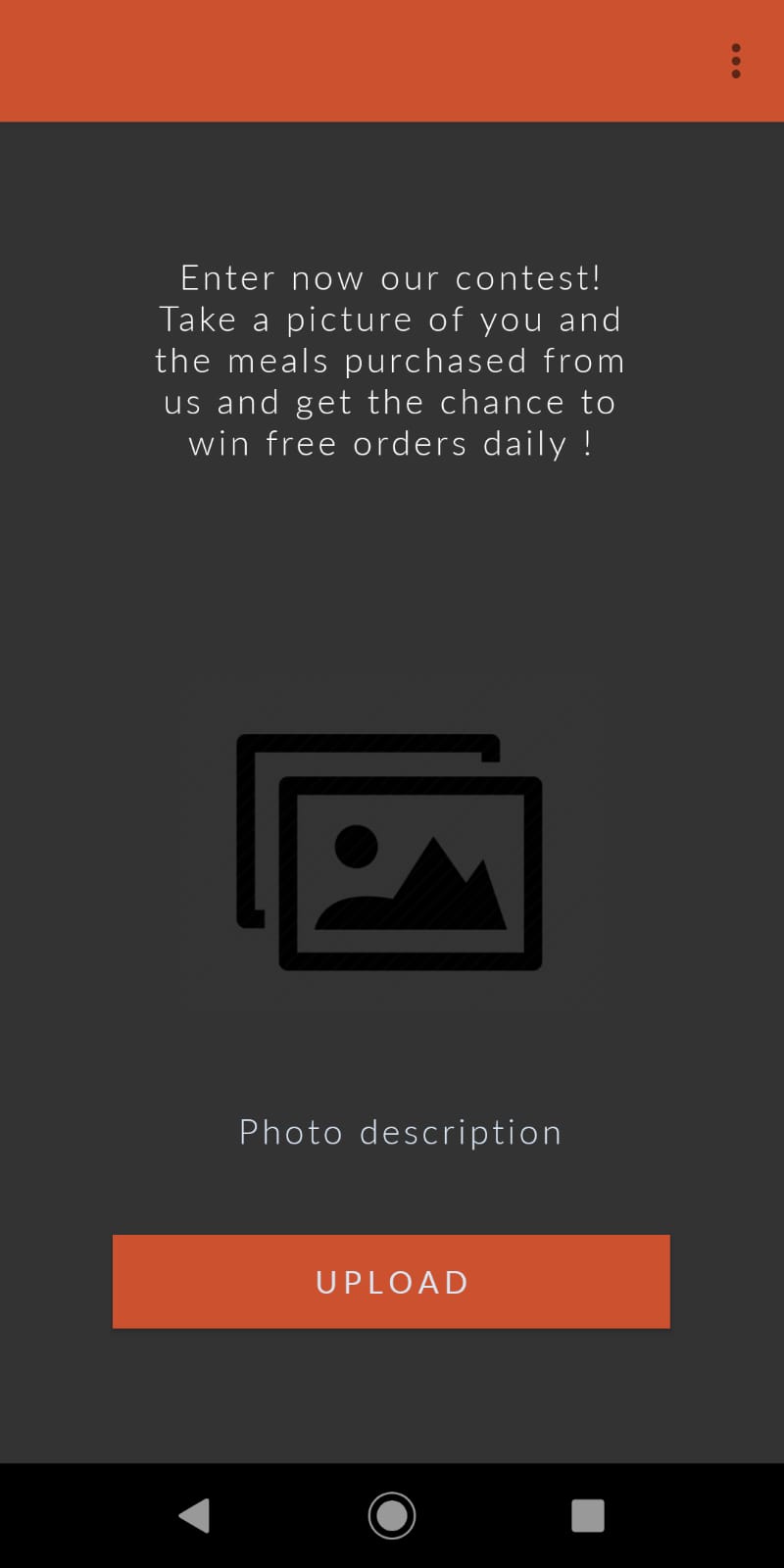
 

Fig. 5.1 Image chosen Fig. 5.1 First view

For the second part, class **UploadImage** is created and it extends AsyncTask<Void,Void, Void> since background operations of a short period of time will be run and the result will be published on the UI thread .It stores the image in Bitmap format and the name as a String(the name is entered by the user through a simple EditText).

The *doInBackground*() method compresses the image from JPEG format into ByteArrayOutputStream (compress() method) so it can afterwards encode it into a String in Base64(encodeToString() method).

Then the data to be sent to the server is created as an ArrayList<NameValuePair> with components set as the encoded format of the image and its name. The transmission process is done through HTTP protocol (HttpPost) and an URL encoded entity of the data to send is sent. This is done by using another php script which handles the next layer to the server.

The php script - **SavePicture**.php – does nothing but to decode the image from the base 64 string and stores the result image as JPG format into ‘pictures’ folder on the server.

The HTTP attributes of the request to the server are also set in the getHttpRequestParams() method which sets the connection and the socket timeout as 30 seconds.

The onPostExecute() method does nothing but to create a Toast message when the image is successfully uploaded on the server .

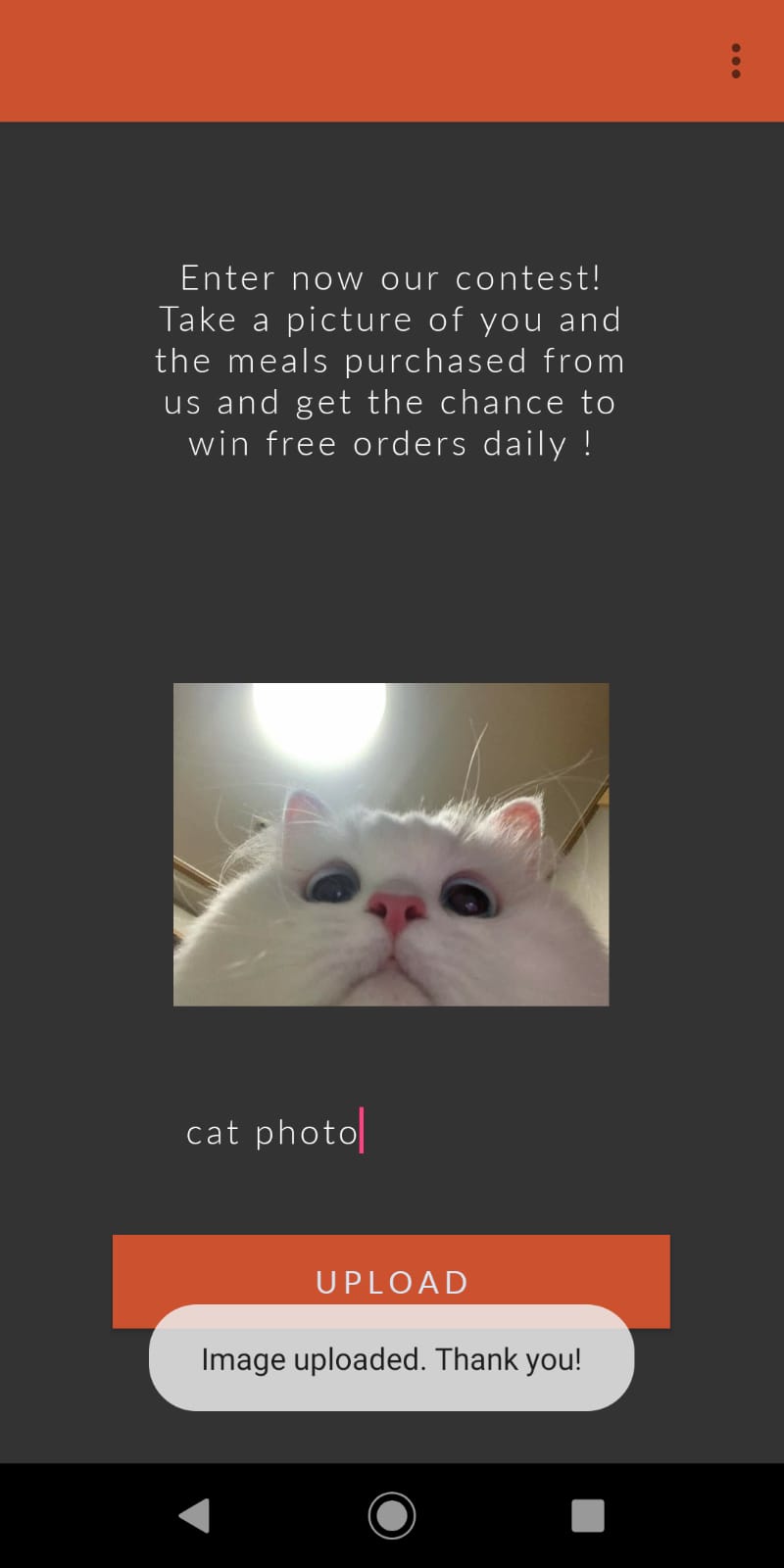


Fig. 5.2 Image uploaded

# 6. Wifi Listener

To monitor the internet connection, the class **NetworkMonitor** is created. The method *registerConnectivityNetworkMonitor*() is called from the onCreate() method of the activities. It creates a ConnectivityManager object and calls registerNetworkCallback() on it with the following parameters :

[NetworkRequest](https://developer.android.com/reference/android/net/NetworkRequest.html) *request* as a NetworkRequest Builder

[ConnectivityManager.NetworkCallback](https://developer.android.com/reference/android/net/ConnectivityManager.NetworkCallback.html) *networkCallback* as a [ConnectivityManager.NetworkCallback](https://developer.android.com/reference/android/net/ConnectivityManager.NetworkCallback.html) which implements the onAvailable() interface method and onLost() interface method as broadcasting the newly created method *getConnectivityIntent*() with the parameter set as ‘false’, respectively ‘true’ .

*getConnectivityIntent() method* sets the intent action to be performed as CONNECTIVITY\_CHANGE and adds extended data to the intent (name as EXTRA\_NO\_CONNECTIVITY and the Parcelable message set as sent through boolean parameter). This way the application is monitored on the *background*.

**Limitations**: This implementation only works for API 21 and up.

On the *foreground*, inside the onCreate() method of the activity, *isConnected*() method is called and if it returns false, then an alert dialog is issued. The function creates a ConnectivityManager and calls getActiveNetworksInfo() on it. If the returned value is not ‘null’ and the isConnectedOrConnecting() method returns ‘true’, then it gets the network info for Mobile Data or WIFI types of Internet network. If one of them is not ‘null’, then we know for sure that the device is connected to the Internet.

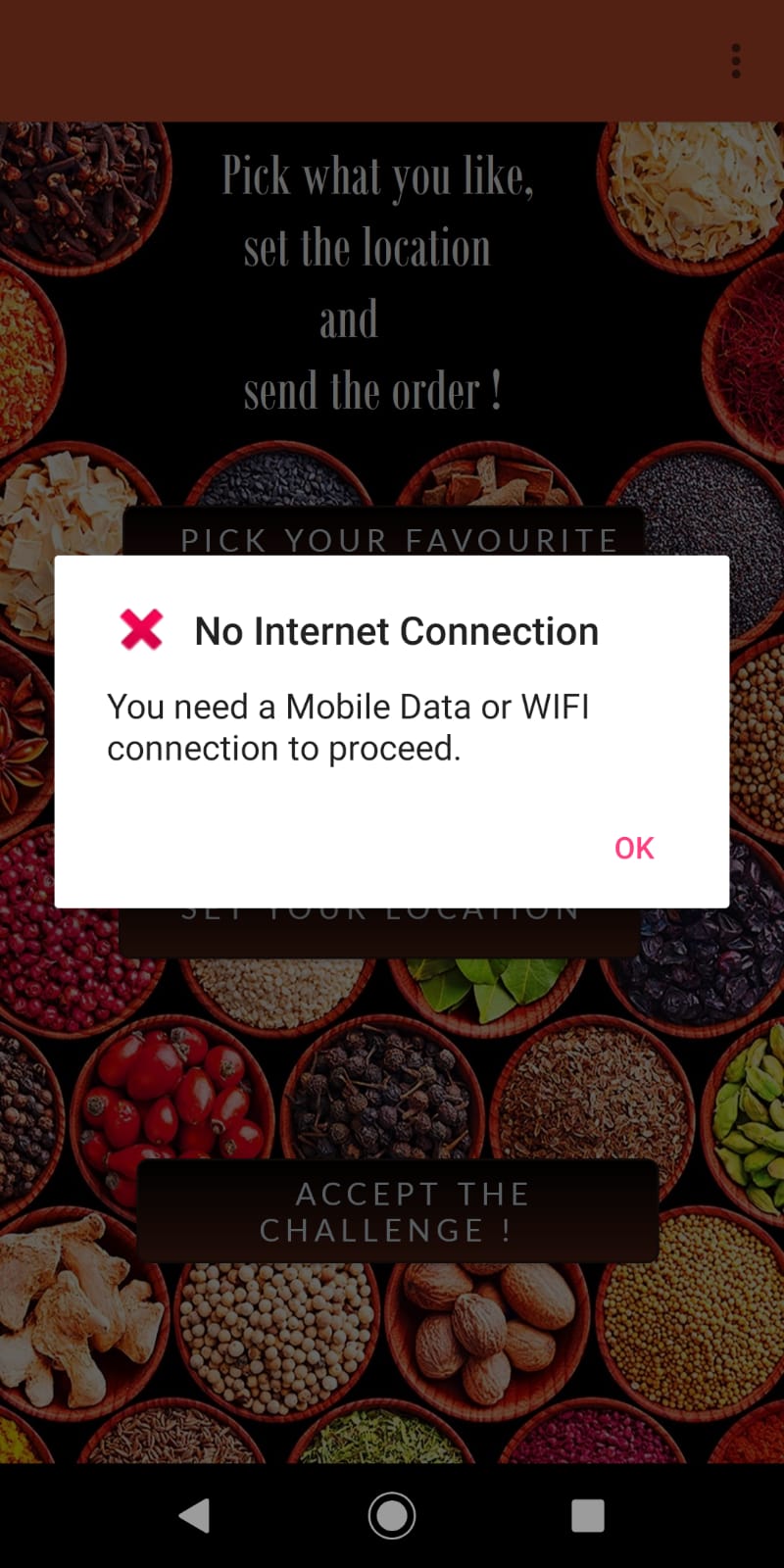


Fig. 6.1 Alert dialog for missing Internet connection

# 7. Notifications

In order for the app to be able to send or receive notifications (push or device to device), the application was linked to the Firebase platform. The application implements a push notifications sample code as follows.

The class **FirebaseMessagingService** extends FirebaseMessagingService and it is where FCM messages are handled.

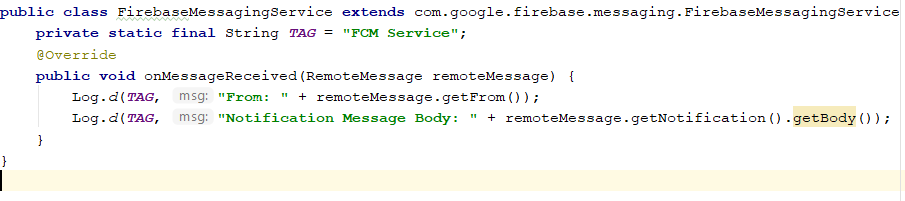


Fig.7.1 FirebaseMessagingService

The class **FirebaseInstanceIDService** extends FirebaseInstanceIdService and it is where to associate the user's FCM InstanceID token with any server-side account maintained by the application.

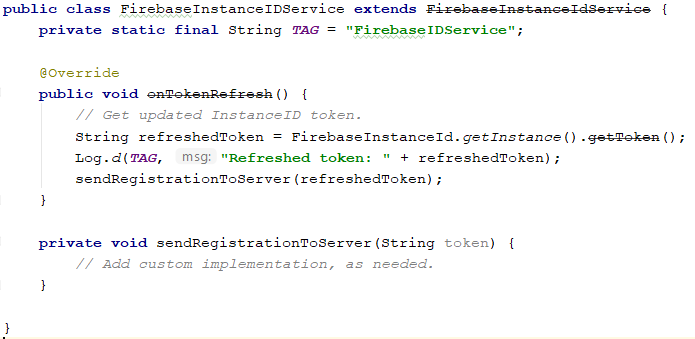


Fig.7.2 FirebaseInstanceIDService

# 8. Maps and Directions

The user is able to search for a specific address on a Google map and get the current location. The address will be used as the delivery address.

**MapMainActivity** performs the checks before the map is created and initializes the actual map activity (**MapActivity**).

First thing to do is check that the device the app is running on has the correct map version.

The method *isServicesOK*() checks the Google services version. If for the current context the Google Play services APK is available and up-to-date on this device (GoogleApiAvailability.getInstance().isGooglePlayServicesAvailable(context)), then the process can continue and map requests can be done. If something goes wrong, but the error is solvable by user then show the Dialog box right from Google (GoogleApiAvailability.getInstance().getErrorDialog) to show where they can go to resolve the error(GoogleApiAvailability.getInstance().isGooglePlayServicesAvailable).

The method *init*() does nothing but to call the **MapActivity** start. It is only called when *isServicesOK*() returns TRUE from onCreate() method.

**MapActivity** implements all the functionality to be performed on the Google map (implements OnMapReadyCallback).

It holds the permissions from the Manifest for Fine and Course Location(*FINE\_LOCATION* and *COURSE\_LOCATION), LOCATION\_PERMISSION\_REQUEST\_CODE* – the code for requesting permissions, *DEFAULT\_ZOOM* value for the camera, a boolean for storing if the location permissions are granted, the GoogleMap map, and a FusedLocationProviderClient.

First, we need to make sure we have all the required permissions to use the map. The method *getLocationPermission*() explicitly checks the permissions(Fine Location and Course Location)and it is called from onCreate()method. The global variables from Manifest File are checked by using checkSelfPermission() and the global boolean variable is set to ‘true’ if permissions are all set .Also, the *initMap*() is called if everything is ok. If they are not, just request the needed permissions by calling requestPermissions() .

Moving over, if a request for location permissions was issued, then we need the result of the request which will be received by overriding the onRequestPermissionResult() method. If the request code was the one defined in this activity and used when the request was made, then the parameter grantResults of this method is checked. If the length is greater than ‘0’ and also if all the elements of grantResults are equal to PERMISSION\_GRANTED, then that means permissions were granted and we can set the global status of permissions to ‘true’ and of course initialize the map by calling *initMap*().

The *initMap*() method creates the map fragment .

The interface method *onMapReady*() creates the map object when the map is ready. After the map was created, we can go on and get the device location by calling *getDeviceLocation*() if the location permissions are granted. A search bar is also added in order to give the user the possibility to search for some other location. But since the search bar placing cannot be modified and it will cover the button used for finding the current location, then we cannot simply call setMyLocationEnabled(true). We need to hide it from the UI and set another customized one (call getUiSetings().setMyLocationEnabled(false)).

The *init*() method configures the search bar to invoke the moving of the camera to the searched address by calling *geoLocate*() method when some text is entered into it. Also, here is where OnClickListener() for the autolocating button is configured to call *getDeviceLocation*() . It is called from *onMapReady*() interface method.

*geoLocate*() method gets the text inputted in the search bar and gets a list of the addresses that is returned by getFromLocationName() of the Geocode object. We will only use the first element of the list and call *moveCamera*() with the chosen address coordinates.

The method *getDeviceLocation*() initializes the Fused Location Provider Client global variable with the result returned by getFusedLocationProviderClient() of LocationServices. The variable which holds the status of the location permissions is stored is checked in order to proceed with the locating process. If the permissions are granted, then call getLastLocation() of the Fused Location Provider Client. If the task was successful (check isSuccesfull() in onComplete() interface method of the task), then we get the result of the task and move the camera on it(call moveCamera()).

*moveCamera*() method gets as parameters the latitude and longitude coordinates and the zooming factor. It calls the *moveCamera*() interface method with the given parameters for latitude, longitude and zoom factor. Moreover, we need to create a marker and add on the map. The title of the marker is also passed through the parameter.

After the search text is inputted, it would be preferable to hide the keyboard since it won’t come down automatically. For this another method is created *hideSoftKeyboard*() which simply calls getWindow().setSoftInputMode(WindowManager.LayoutParams.SOFT\_INPUT\_STATE\_ALWAYS\_HIDDEN) and it is called at the end of *init*().

In order to be able to use Google map services, API Key was created to correspond to the package of the application and configurated the app in order to use the generated key. Also, Google Maps Android API was enabled.

Fig.8.2 Result on search on map Fig.8.1 Map View

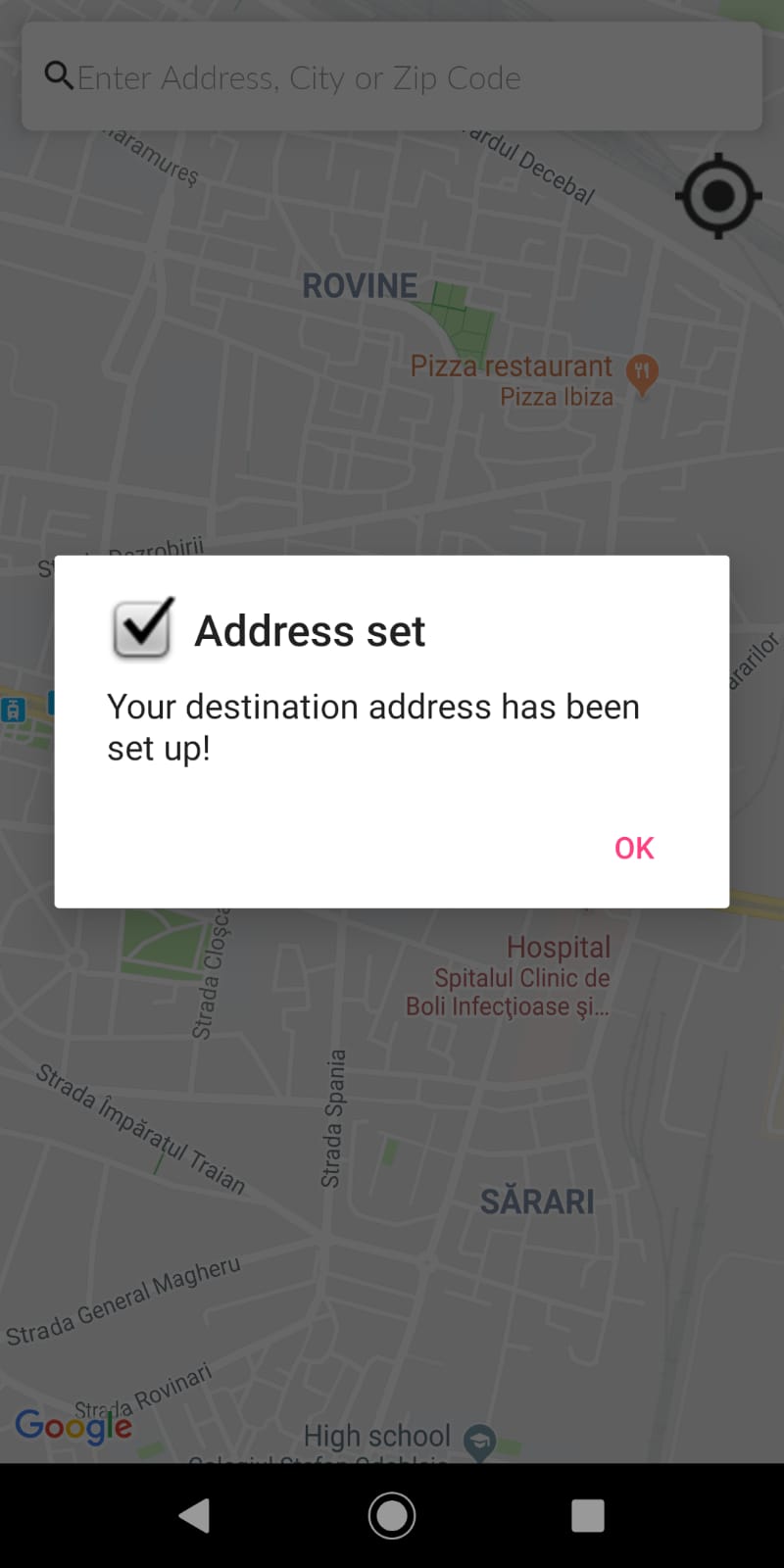
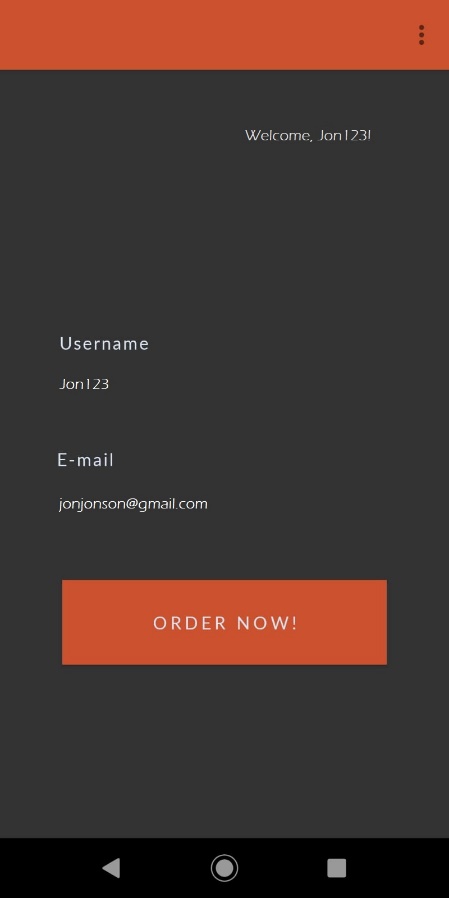
 

Fig.8.3 Result of location getting Fig.8.4 Address set

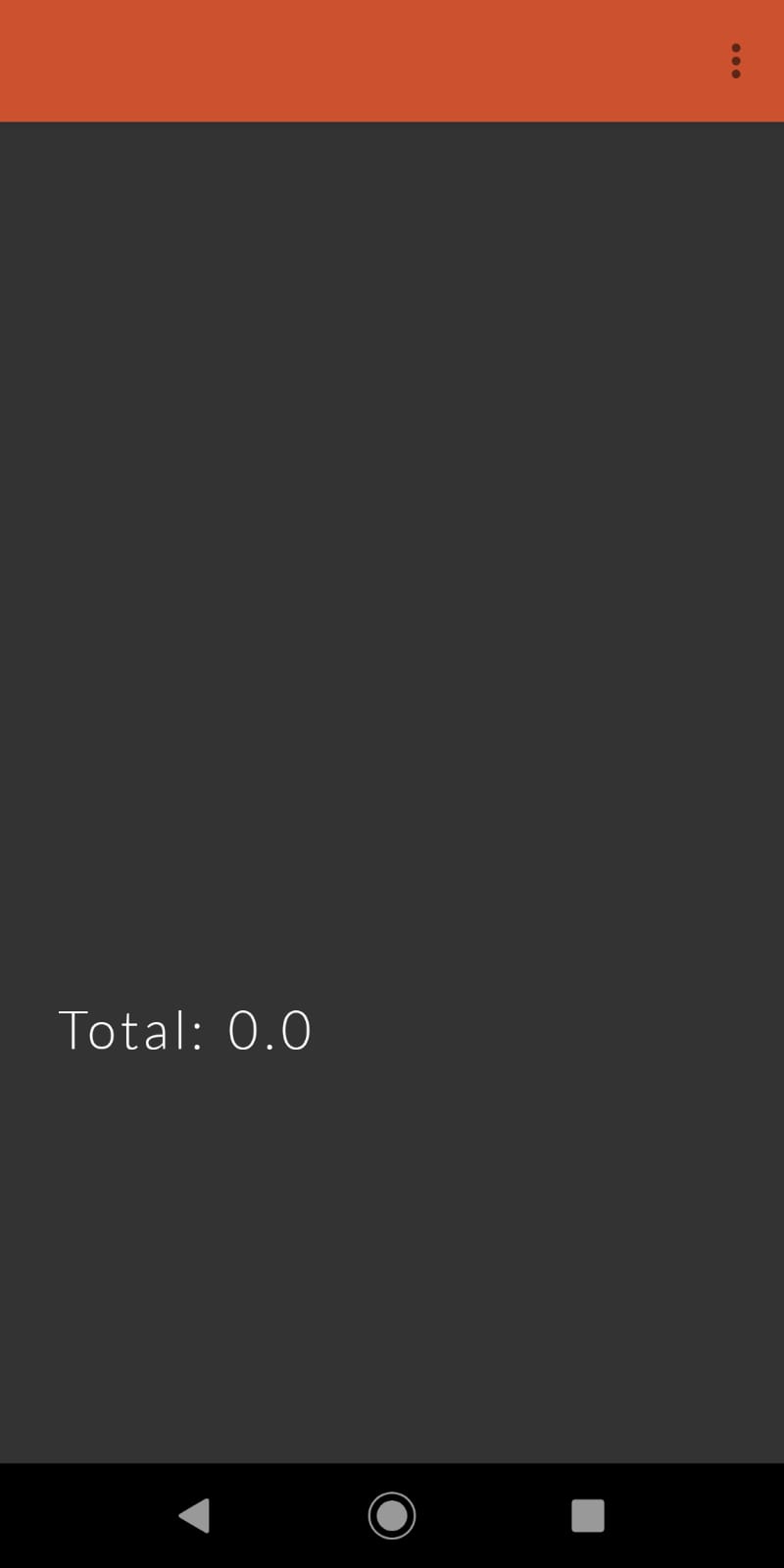


Fig.8.5 Main Map View

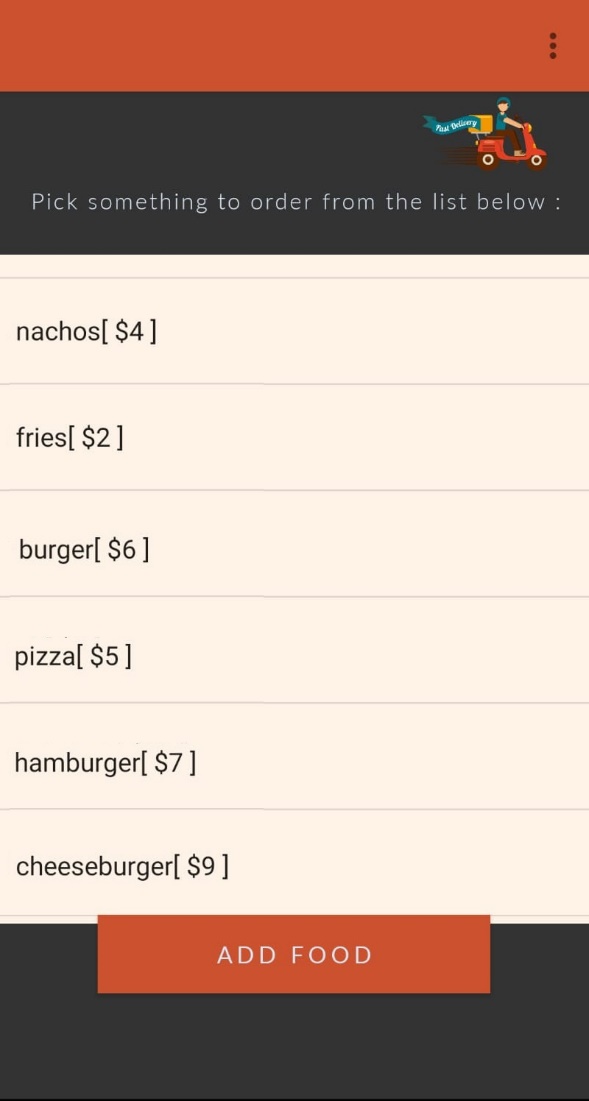
* **Additional screenshots**

Main view User Area View

b 

Order view Empty Order view

User Foods View Admin Foods View