**TABLE OF CONTENTS**

**ABSTRACT 1**

**CHAPTER 1: INTRODUCTION 2-5**

* 1. **INTRODUCTION**
  2. **SCOPE OF THE PROJECT**
  3. **FUNCTIONAL REQUIREMENT**
  4. **NON-FUNCTIONAL REQUIREMENT**
  5. **SOFTWARE REQUIREMENT**
  6. **HARDWARE REQUIREMENT**

**CHAPTER 2: DESIGN 6-10**

**2.1 ER DIAGRAM**

**2.2 ACTIVITY DIAGRAM**

**2.3 SEQUENCE DIAGRAM**

**2.4 USECASE DIAGRAM**

**2.5 CLASS DIAGRAM**

**CHAPTER 3: OBJECTIVE 11**

**3.1 OBJECTIVE**

**3.2 DESCRIPTION**

**CHAPTER 4: PROJECT CODE 12-29**

**4.1 PROJECT CODE**

**4.2 SCREENSHORTS**

**4.3 TESTCASES**

**CHAPTER 5: CONCLUSION 30-31**

**5.1 CONCLUSION**

**5.2 REFERENCES**

**ABSTRACT**

The food service industry is fast-paced, demanding efficiency and precision. One key aspect of restaurant operations is the timely delivery of food from the kitchen to the customers' tables. To enhance this process, we propose the development of a Food Runner Application. This application aims to optimize food running tasks by providing real-time communication and task management capabilities.

The Food Runner Application will enable seamless coordination between kitchen staff, servers, and food runners. Utilizing a user-friendly interface, it will allow servers to input food orders and assign them to specific food runners. Food runners will receive instant notifications regarding assigned tasks, including order details and table numbers, ensuring prompt and accurate food delivery.

Moreover, the application will include features such as route optimization, enabling food runners to efficiently navigate through the restaurant floor. Real-time updates on order statuses and table availability will further enhance operational transparency and customer satisfaction.

By leveraging modern technology and prioritizing user experience, the Food Runner Application seeks to revolutionize food delivery processes in restaurants. Its implementation promises to streamline operations, reduce wait times, and ultimately elevate the dining experience for patrons and staff alike.

**CHAPTER 1**

**1.1 INTRODUCTION**

In a fast-paced world where convenience and efficiency are paramount, the need for seamless food delivery solutions has never been greater. Enter our Food Runner App, the ultimate tool designed to revolutionize the way food is delivered from restaurants to hungry customers.

Our Food Runner App serves as the bridge between restaurants and customers, offering a streamlined platform that connects users with their favorite eateries and ensures prompt delivery of delicious meals right to their doorstep. Whether it's a busy professional craving a quick lunch, a family seeking a convenient dinner option, or a group of friends hosting a gathering, our app caters to all.

At its core, our Food Runner App is more than just a delivery service—it's a testament to the power of technology to enhance everyday experiences and bring people together over great food. Whether you're a hungry customer craving a tasty meal or a restaurant looking to reach new heights, our app is here to make it happen. Join us on this culinary journey and let's redefine the future of food delivery, one order at a time.

**1.2 SCOPE:**

The scope for a food runner app is substantial, given the increasing demand for convenient food delivery services. Such an app would streamline the process of connecting customers with nearby restaurants, facilitating quick and efficient delivery of meals. With the rise of busy lifestyles and the preference for online ordering, there's a significant market opportunity for a food runner app to thrive. Moreover, partnerships with local eateries and efficient logistics management can further optimize the delivery process, ensuring timely and accurate deliveries. Overall, the scope for a food runner app is promising, provided it delivers on convenience, reliability, and quality service.

**1.3 FUNCTIONAL REQUIREMENTS:**

1. Allow users to register and create accounts.

2. Provide secure authentication mechanisms (e.g., email/password, social login) to verify user identities.

3. Display a menu of available food items with detailed descriptions, prices, and images to facilitate order selection.

4. Enable users to view their order history..

**1.4 NON FUNCTIONAL REQUIREMENTS:**

1. The app will load quickly, with minimal latency.

2. The user interface is intuitive and easy to navigate for customers.

3. The app is optimized to minimize battery consumption, ensuring prolonged usage without draining the device's battery excessively.

4. All communication between the app and backend servers should be encrypted to prevent data interception or tampering.

**1.5 SOFTWARE REQUIREMNETS:**

* Programming Language: Kotlin, XML.
* IDE : Android Studio.
* Data Base: SQL, PHP.
* Operating System: Windows 7 or above.

**1.6 HARDWARE REQUIREMENTS:**

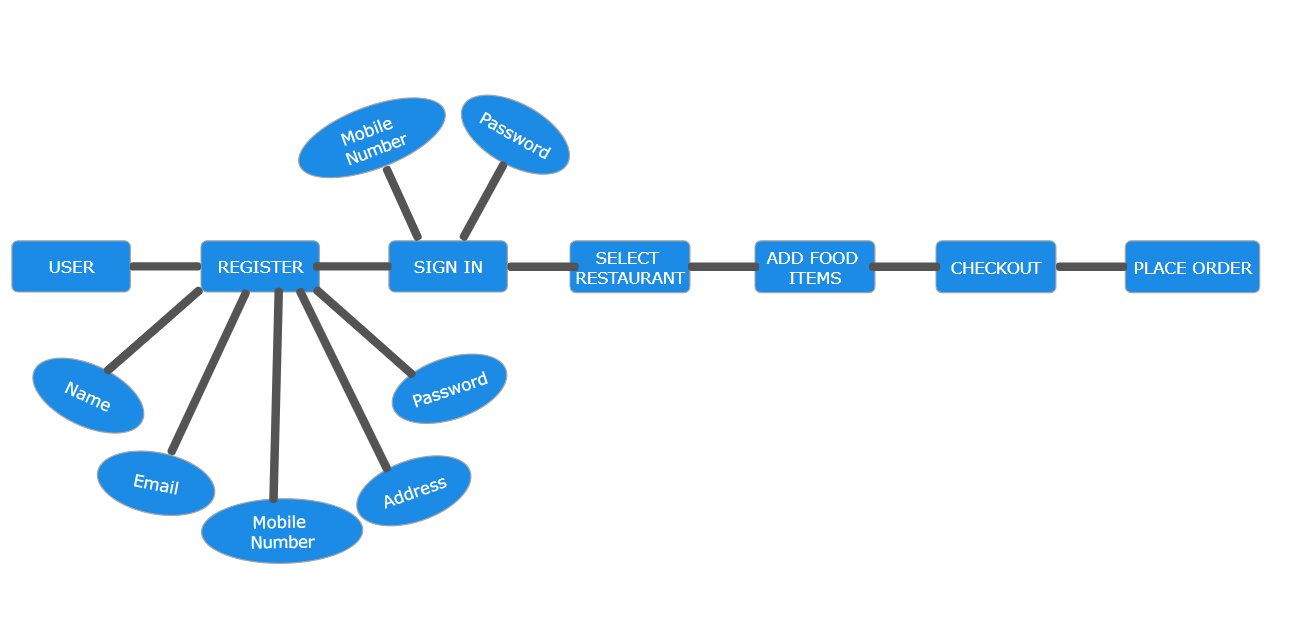
* RAM: 4GB (further increase that as per requirement.)
* Storage: 2GB (further increase that as per requirement.)
* **Screen Resolution:** 1280 x 800
* Minimum required JDK Version: Java Development Kit 8.

**CHAPTER -2**

**DESIGN**

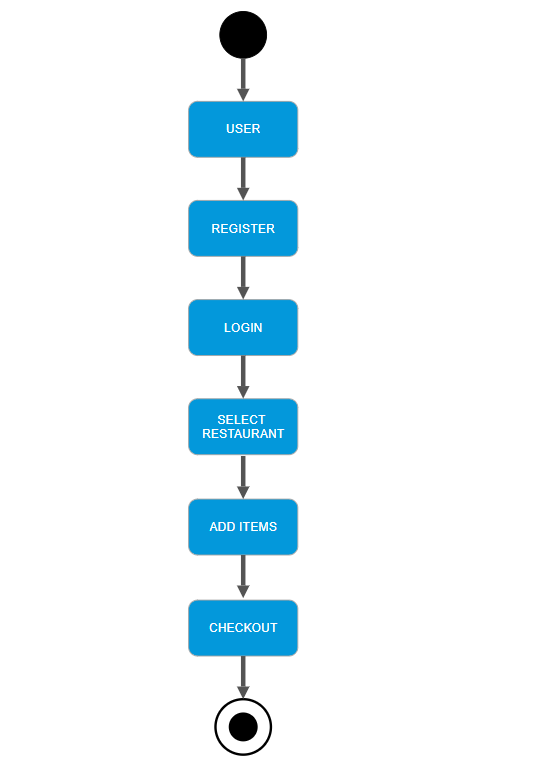
**2.1 ER DIAGRAM:**

An Entity-Relationship (ER) diagram is a visual representation of the data model that depicts the logical structure of databases. It's composed of entities (which are usually nouns representing objects or concepts) and the relationships between them.

****

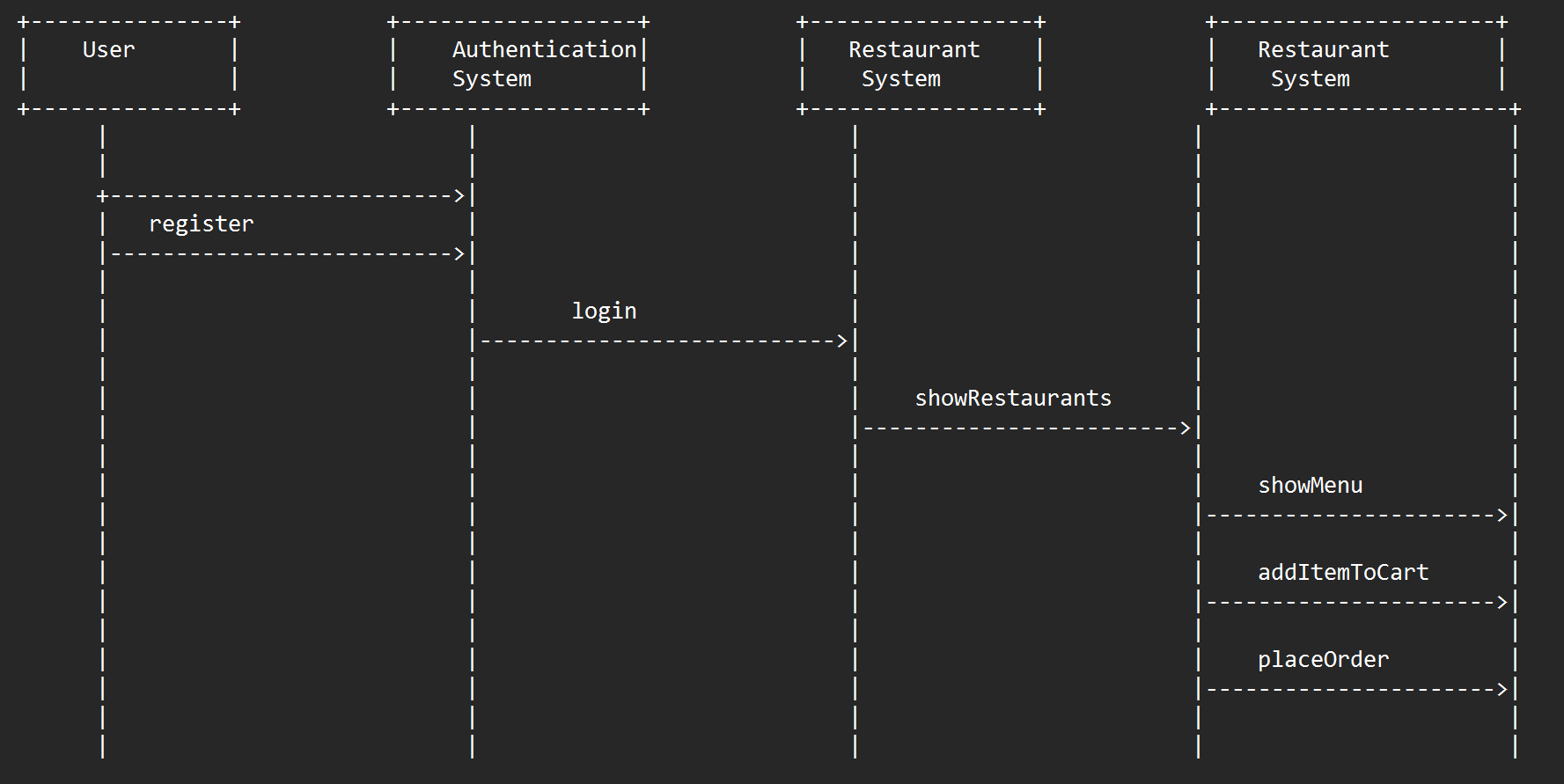
**2.2 ACTIVITY DIAGRAM:**

The process flows in the system are captured in the activity diagram. Similar to a state diagram, an activity diagram also consists of activities, actions, transitions, initial and final states, and guard conditions.

****

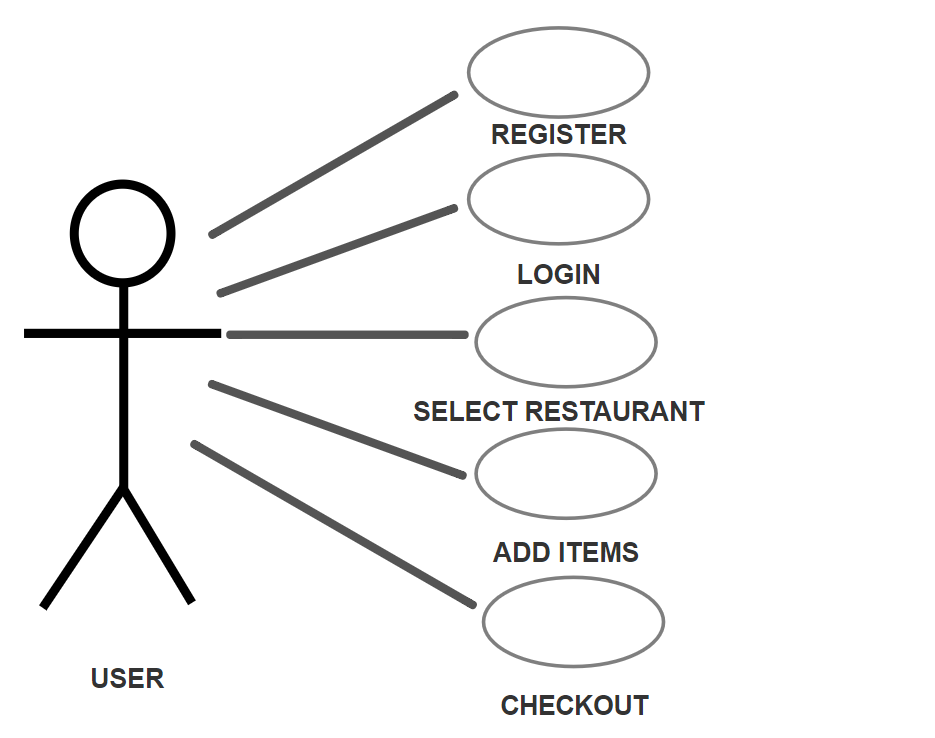
**2.3 SEQUENCE DIAGRAM:**

A sequence diagram represents the interaction between different objects in the system. The important aspect of a sequence diagram is that it is time-ordered. This means that the exact sequence of the interactions between the objects is represented step by step. Different objects in the sequence diagram interact with each other by passing "messages".

****

**2.4 USECASE DIAGRAM:**

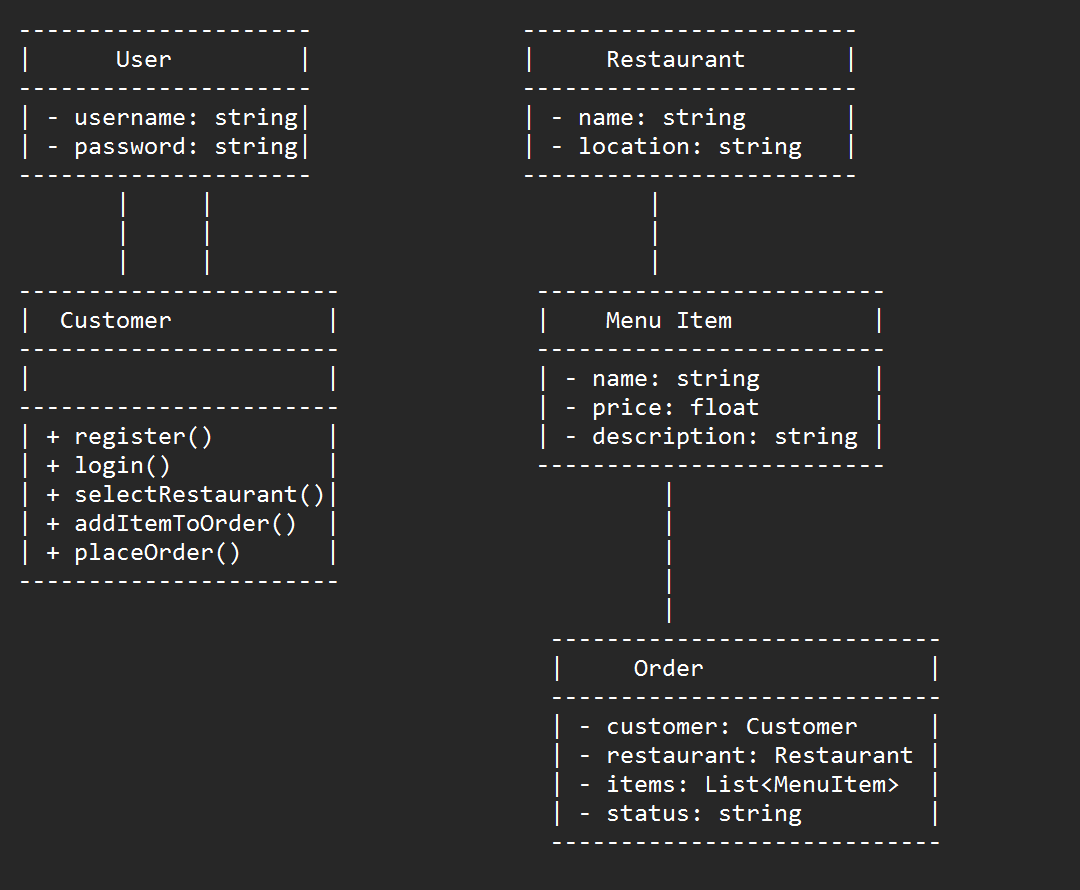
The use case diagram is used to identify the primary elements and processes that form the system. The primary elements are termed as "actors" and the processes are called "use cases." The use case diagram shows which actors interact with each use case.

****

**2.5 CLASS DIAGRAM:**

The class diagram is used to refine the use case diagram and define a detailed design of the system. The class diagram classifies the actors defined in the use case diagram into a set of interrelated classes. The relationship or association between the classes can be either an "is-a"

or "has-a" relationship. Each class in the class diagram may be capable of providing certain functionalities. These functionalities provided by the class are termed "methods" of the class. Apart from this, each class may have certain "attributes" that uniquely identify the class.

****

**CHAPTER 3**

**3.1 OBJECTIVE:**

To create a seamless and efficient platform that connects restaurants with food runners, enabling timely and accurate delivery of orders to customers while optimizing operational workflows for both businesses and delivery personnel. To revolutionize dining experiences by creating a seamless and efficient platform that connects restaurants with food runners, ensuring timely and accurate delivery of orders while enhancing customer satisfaction and optimizing restaurant operations. To streamline the food ordering and delivery process for both users and restaurants. To provide a convenient and user-friendly platform for discovering and ordering food from a variety of restaurants. To connect restaurants with a network of reliable and efficient food runners for timely deliveries. To enhance the overall dining experience by offering real-time order tracking and communication features.

**3.2 DESCRIPTION:**

Introducing the ultimate solution for seamless food delivery - the Food Runner app. Whether you're a busy professional craving your favorite dish, a parent looking for a hassle-free dinner option, or a foodie exploring new culinary delights, Food Runner is your go-to companion for satisfying your cravings at the tap of a button.

With Food Runner, you can browse through a diverse range of restaurants, cafes, and eateries in your area, all conveniently sorted based on your preferences. From mouthwatering burgers to authentic ethnic cuisine, we've got it all covered. Simply choose your desired dishes, customize your order with special instructions, and let Food Runner handle the rest.

What sets Food Runner apart is its commitment to efficiency and reliability. Our advanced algorithm ensures that your order is swiftly processed and assigned to the nearest available runner, guaranteeing prompt delivery straight to your doorstep. Plus, real-time tracking lets you monitor the progress of your order every step of the way, giving you peace of mind and eliminating any guesswork.

Safety and security are paramount at Food Runner. Rest assured that every runner undergoes a thorough screening process and adheres to strict hygiene protocols, ensuring that your food arrives fresh, hot, and untouched. We also offer contactless delivery options for added convenience and peace of mind.

**CHAPTER 4**

**4.1 PROJECT CODE:**

package com.anujandankit.foodrunner.activity

import android.content.Context

import android.content.Intent

import android.os.AsyncTask

import android.os.Bundle

import android.view.View

import android.widget.ProgressBar

import android.widget.Toast

import androidx.appcompat.app.AppCompatActivity

import androidx.recyclerview.widget.LinearLayoutManager

import androidx.recyclerview.widget.RecyclerView

import androidx.room.Room

import com.android.volley.Response

import com.android.volley.toolbox.JsonObjectRequest

import com.android.volley.toolbox.Volley

import com.anujandankit.foodrunner.R

import com.anujandankit.foodrunner.adapter.MenuItemRecyclerViewAdapter

import com.anujandankit.foodrunner.database.MenuItemEntity

import com.anujandankit.foodrunner.database.OrderEntity

import com.anujandankit.foodrunner.database.RestaurantDatabase

import com.anujandankit.foodrunner.database.RestaurantEntity

import com.anujandankit.foodrunner.fragment.HomeFragment

import com.anujandankit.foodrunner.model.MenuItem

import com.anujandankit.foodrunner.util.ConnectionManager

import com.google.android.material.snackbar.Snackbar

import com.google.gson.Gson

import kotlinx.android.synthetic.main.activity\_description.\*

import org.json.JSONException

class DescriptionActivity : AppCompatActivity() {

var restaurantId: Int? = 2901

var restaurantName: String? = ""

var restaurantRating: String? = ""

var restaurantImageUrl: String? = ""

var restaurantCostForOne: String? = ""

lateinit var recyclerView: RecyclerView

lateinit var layoutManager: LinearLayoutManager

lateinit var recyclerAdapter: MenuItemRecyclerViewAdapter

lateinit var progressBar: ProgressBar

val menuItemList = arrayListOf<MenuItem>()

private var orderList = arrayListOf<MenuItem>()

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

setContentView(R.layout.activity\_description)

supportActionBar?.setDisplayHomeAsUpEnabled(true)

proceed\_button.visibility = View.GONE

if (intent != null) {

restaurantId = intent.getIntExtra("restaurant\_id", 2901)

restaurantName = intent.getStringExtra("restaurant\_name")

restaurantRating = intent.getStringExtra("restaurant\_rating")

restaurantCostForOne = intent.getStringExtra("restaurant\_cost\_for\_one")

restaurantImageUrl = intent.getStringExtra("restaurant\_image\_url")

supportActionBar?.title = restaurantName

} else {

finish()

Toast.makeText(

this@DescriptionActivity,

"Some Unexpected error occurred",

Toast.LENGTH\_SHORT

).show()

}

if (restaurantId == 2901) {

finish()

Toast.makeText(

this@DescriptionActivity,

"Some Unexpected error occurred",

Toast.LENGTH\_SHORT

).show()

}

proceed\_button.setOnClickListener {

/\*Here we see the implementation of Gson.

\* Whenever we want to convert the custom data types into simple data types

\* which can be transferred across for utility purposes, we will use Gson\*/

val gson = Gson()

/\*With the below code, we convert the list of order items into simple string which can be easily stored in DB\*/

val foodItems = gson.toJson(orderList)

val async =

ItemsOfCart(

this@DescriptionActivity,

restaurantId.toString(),

foodItems,

1

).execute()

val result = async.get()

if (result) {

val data = Bundle()

data.putInt("resId", restaurantId as Int)

data.putString("resName", restaurantName)

val intent = Intent(this@DescriptionActivity, CartActivity::class.java)

intent.putExtra("data", data)

startActivity(intent)

} else {

Toast.makeText(

this@DescriptionActivity,

"Some unexpected error",

Toast.LENGTH\_SHORT

).show()

}

}

recyclerView = findViewById(R.id.recycler\_view\_description)

layoutManager = LinearLayoutManager(this@DescriptionActivity)

progressBar = findViewById(R.id.progress\_circular\_description)

val queue = Volley.newRequestQueue(this@DescriptionActivity)

val url = "http://13.235.250.119/v2/restaurants/fetch\_result/$restaurantId"

if (ConnectionManager().checkConnectivity(this@DescriptionActivity)) {

val jsonObjectRequest =

object : JsonObjectRequest(

Method.GET, url,

null,

Response.Listener {

try {

val dataObject = it.getJSONObject("data")

val success = dataObject.getBoolean("success")

if (success) {

progressBar.visibility = View.GONE

val data = dataObject.getJSONArray("data")

for (i in 0 until data.length()) {

val menuItemJSONObject = data.getJSONObject(i)

val menuItemObject = MenuItem(

menuItemJSONObject.getString("id"),

menuItemJSONObject.getString("name"),

menuItemJSONObject.getString("cost\_for\_one"),

menuItemJSONObject.getString("restaurant\_id")

)

menuItemList.add(menuItemObject)

recyclerAdapter =

MenuItemRecyclerViewAdapter(

this@DescriptionActivity,

menuItemList,

object :

MenuItemRecyclerViewAdapter.OnItemClickListener {

override fun onAddItemClick(menuItem: MenuItem) {

orderList.add(menuItem)

if (orderList.isNotEmpty()) {

proceed\_button.visibility = View.VISIBLE

}

}

override fun onRemoveItemClick(menuItem: MenuItem) {

orderList.remove(menuItem)

if (orderList.isEmpty()) {

proceed\_button.visibility = View.GONE

}

}

}

)

recyclerView.adapter = recyclerAdapter

recyclerView.layoutManager = layoutManager

}

} else {

val errorMessage = dataObject.getString("errorMessage")

progressBar.visibility = View.GONE

Snackbar.make(

coordinatorViewDescription,

errorMessage,

Snackbar.LENGTH\_INDEFINITE

).show()

}

} catch (e: JSONException) {

progressBar.visibility = View.GONE

Snackbar.make(

coordinatorViewDescription,

"Some unexpected error has occurred while we were handling the data.",

Snackbar.LENGTH\_INDEFINITE

).show()

}

},

Response.ErrorListener {

progressBar.visibility = View.GONE

Snackbar.make(

coordinatorViewDescription,

"We failed to fetch the data. Please Retry.",

Snackbar.LENGTH\_INDEFINITE

).show()

}) {

override fun getHeaders(): MutableMap<String, String> {

val headers = HashMap<String, String>()

headers["Content-type"] = "application/json"

headers["token"] = "2d020a6c927f14"

return headers

}

}

queue.add(jsonObjectRequest)

} else {

Toast.makeText(

this@DescriptionActivity,

"No Internet Connection Found.",

Toast.LENGTH\_SHORT

).show()

finish()

}

val restaurantEntity = RestaurantEntity(

restaurantId.toString(),

restaurantName.toString(),

restaurantRating.toString(),

restaurantCostForOne.toString(),

restaurantImageUrl.toString()

)

val checkFav =

HomeFragment.DBAsyncTask(this@DescriptionActivity, restaurantEntity, 1).execute()

val isFav = checkFav.get()

if (isFav) {

imgFavoriteImageDescription.setImageResource(R.drawable.ic\_favorite\_colored\_24dp)

} else {

imgFavoriteImageDescription.setImageResource(R.drawable.ic\_favorite\_outlined\_colored\_24dp)

}

imgFavoriteImageDescription.setOnClickListener {

if (!HomeFragment.DBAsyncTask(this@DescriptionActivity, restaurantEntity, 1)

.execute()

.get()

) {

val async =

HomeFragment.DBAsyncTask(this@DescriptionActivity, restaurantEntity, 2)

.execute()

val result = async.get()

if (result) {

Toast.makeText(

this@DescriptionActivity,

"Restaurant Added To Favorites",

Toast.LENGTH\_SHORT

).show()

imgFavoriteImageDescription.setImageResource(R.drawable.ic\_favorite\_colored\_24dp)

} else {

Toast.makeText(

this@DescriptionActivity,

"Some Error Occurred",

Toast.LENGTH\_SHORT

).show()

}

} else {

val async =

HomeFragment.DBAsyncTask(this@DescriptionActivity, restaurantEntity, 3)

.execute()

val result = async.get()

if (result) {

Toast.makeText(

this@DescriptionActivity,

"Restaurant Removed From Favorites",

Toast.LENGTH\_SHORT

).show()

imgFavoriteImageDescription.setImageResource(R.drawable.ic\_favorite\_outlined\_colored\_24dp)

} else {

Toast.makeText(

this@DescriptionActivity,

"Some Error Occurred",

Toast.LENGTH\_SHORT

).show()

}

}

}

}

override fun onBackPressed() {

DeleteMenuItems(this@DescriptionActivity).execute()

super.onBackPressed()

}

class DBAsyncTask(

val context: Context,

private val menuItemEntity: MenuItemEntity,

private val mode: Int

) :

AsyncTask<Void, Void, Boolean>() {

/\*

Mode 1- > Item Added or Not

Mode 2 -> Add to cart

Mode 3 -> Remove from cart

\*/

private val db =

Room.databaseBuilder(context, RestaurantDatabase::class.java, "menu-items-db").build()

override fun doInBackground(vararg params: Void?): Boolean {

when (mode) {

1 -> {

val restaurant: MenuItemEntity? = db.menuItemDao()

.getMenuItemById(menuItemEntity.menuItemId)

db.close()

return restaurant != null

}

2 -> {

db.menuItemDao().insertMenuItem(menuItemEntity)

db.close()

return true

}

3 -> {

db.menuItemDao().deleteMenuItem(menuItemEntity)

db.close()

return true

}

}

return false

}

}

class ItemsOfCart(

context: Context,

val restaurantId: String,

private val foodItems: String,

private val mode: Int

) : AsyncTask<Void, Void, Boolean>() {

val db = Room.databaseBuilder(context, RestaurantDatabase::class.java, "res-db").build()

/\* 1-> Insert Order

2-> Delete Order

\*/

override fun doInBackground(vararg params: Void?): Boolean {

when (mode) {

1 -> {

db.orderDao().insertOrder(OrderEntity(restaurantId, foodItems))

db.close()

return true

}

2 -> {

db.orderDao().deleteOrder(OrderEntity(restaurantId, foodItems))

db.close()

return true

}

}

return false

}

}

class DeleteMenuItems(val context: Context) :

AsyncTask<Void, Void, Void>() {

override fun doInBackground(vararg params: Void?): Void? {

val db =

Room.databaseBuilder(context, RestaurantDatabase::class.java, "menu-items-db")

.build()

db.menuItemDao().deleteAllMenuItems()

db.close()

return null

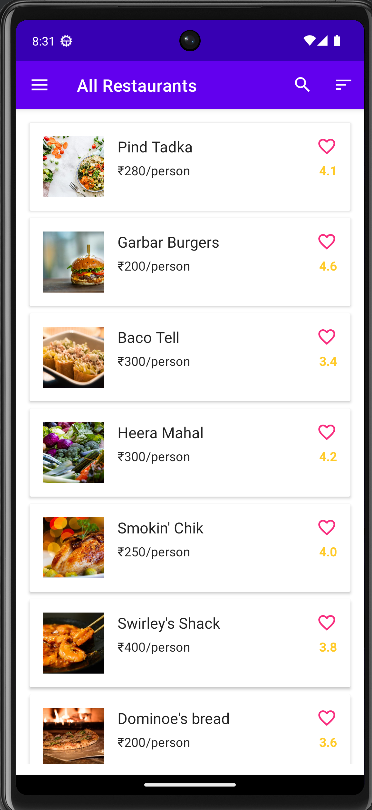
}

}

}

**4.2 SCRREN SHOTS:**

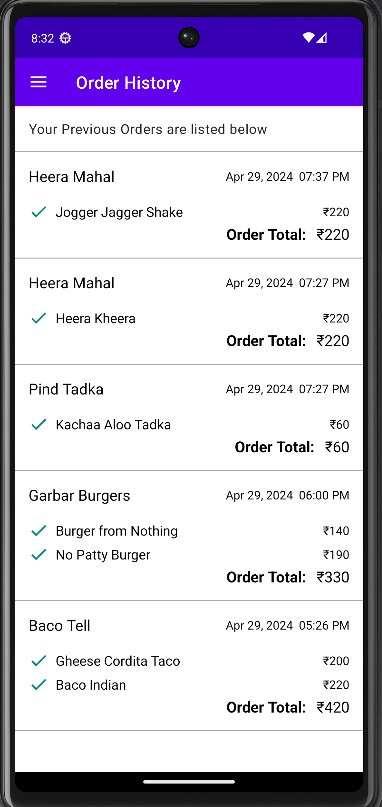
1. **Homepage:**

****

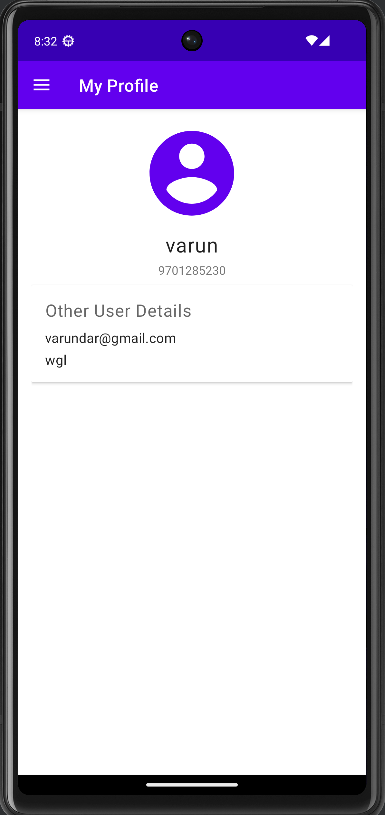
1. **Favorites Page:**



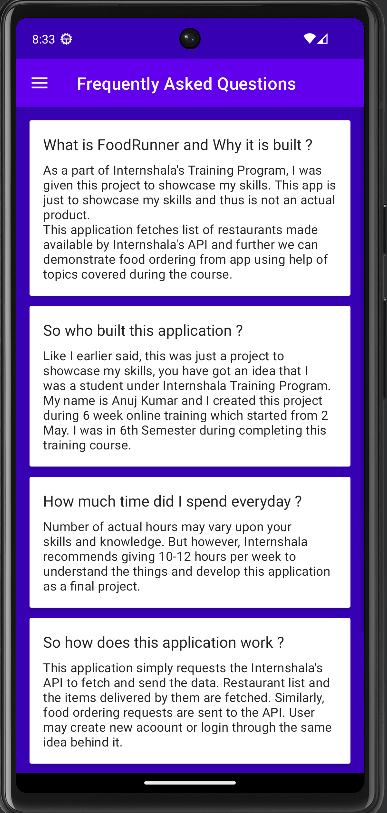
1. **My Orders Page:**



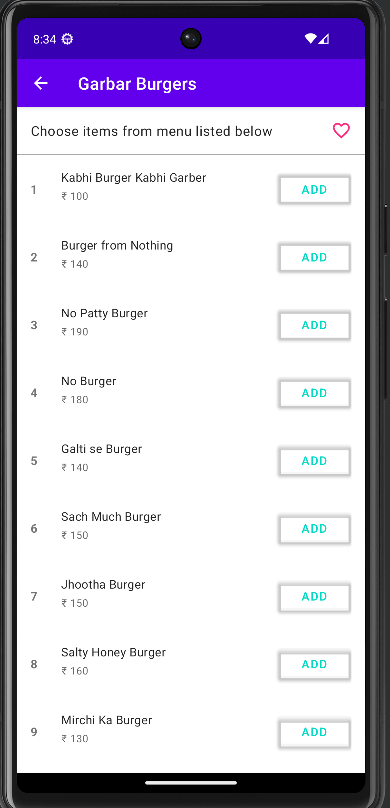
**4. About Me Page:**



**5. FAQ Page:**



**6. Restaurant Item Page:**



**7. My Cart Page:**



**4.3 TEST CASES:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test Case ID | Description | Actual Output | Expected Output | Result |
| 01 | Invalid Login Credentials | Incorrect Mobile Number. | Incorrect Mobile Number. | PASS |
| 02 | Enter Correct Password. | Error: Weak Password | Error: Weak Password | PASS |
| 03 | Add a new item to the menu with valid details | Success message: "Item added successfully." | Success message: "Item added successfully." | PASS |
| 04 | Customer views their order history | List of previous orders displayed | List of previous orders displayed | PASS |

**CHAPTER 5**

**5.1 CONCLUSION:**

In conclusion, the food runner app presents a comprehensive solution to streamline food delivery operations, enhancing efficiency and customer satisfaction. By leveraging innovative technology, the app facilitates seamless communication between restaurants, delivery drivers, and customers, ensuring timely delivery and order accuracy. With its user-friendly interface and robust features, including real-time tracking and secure payment options, the app offers convenience and reliability to all stakeholders involved. As the food delivery market continues to grow, embracing such technological advancements becomes imperative for businesses to stay competitive and meet evolving consumer demands. The food runner app stands as a testament to the power of innovation in revolutionizing the food delivery industry, paving the way for enhanced experiences and greater operational success.

**5.2 REFERENCES:**

1. **UberEats API:** UberEats provides an API that allows developers to integrate their platform into third-party apps. You can find documentation and resources on their developer portal: UberEats Developer.
2. **Postmates API:** Postmates offers an API for developers to build on-demand delivery into their applications. You can explore their API documentation and resources here: Postmates Developer.
3. **Just Eat:** Just Eat is a food delivery service that partners with local restaurants to offer delivery to users. It provides a user-friendly platform for browsing menus, placing orders, and tracking deliveries.
4. **Zomato:** Zomato offers food delivery services as well as restaurant discovery and reviews. It provides users with information about restaurants, menus, and reviews to help them make informed decisions.