PROJECT REPORT ON

OrderEats – Ecommerce Application

Submitted in partial fulfillment of the requirements for the award of the degree of

BACHELOR OF COMPUTER APPLICATIONS

Guru Gobind Singh Indraprastha University, Delhi

Batch: 2023 – 26

Under the Guidance of

Ms. Charanpreet Kaur

Designation

(Associate Professor)

Submitted by:

Rajender Mohan Verma – 35628402023

Shekhar Maurya – 01228402023

Anushka Singh - 35528402023

BOSCO TECHNICAL TRAINING SOCIETY

(Affiliated to GGSIP University, Delhi)

Don Bosco Technical School, Okhla Road, New Delhi 110025

9643868820, 8527787221, 011-41033889

Declaration

We hereby declare that the project report entitled "OrderEats – Food Ordering Android App" is a record of original work carried out by us during our training period. The work reported here has not been submitted to any other institute or university for the award of any degree or diploma. The project was executed under the guidance of our mentor and follows the academic rules and ethical practices expected from student projects.

Certified that the Project Report submitted in partial fulfillment of Bachelor of Computer Applications (BCA) to be awarded by G.G.S.J.P. University, Delhi by Rajender Mohan Verma, Enrolment No. 35628402019, Shekhar Maurya, Enrolment No. 01228402019, by Anushka Singh, Enrolment No. 35528402019 has been completed under my guidance and is Satisfactory

Date -

Place – New Delhi

Signature of the Faculty/Guide

Ms. Charanpreet Kaur

Name of the Faculty /Guide

Associate Professor

Designation

Acknowledgement

First and foremost, we express our sincere gratitude to the Almighty for giving us the strength, dedication, and determination to successfully complete our project *"OrderEats – A Food Ordering Application"*.

We are deeply thankful to our respected mentor **Ms. Charanpreet Kaur** for her constant guidance, encouragement, and valuable suggestions throughout the project. Her support, expert knowledge, and constructive feedback have been a major source of inspiration, helping us overcome challenges and complete the project successfully.

We also extend our heartfelt thanks to our parents and friends for their continuous motivation, encouragement, and moral support during the course of this work.

Finally, we are proud to present this project, which reflects our teamwork, dedication, and learning.

Team Members:

- Rajender Mohan Verma – Enroll. No: 35628402023

  Signatures: _____
- Shekhar Maurya – Enroll. No: 01228402023

  Signatures: _____
- Anushka Singh – Enroll. No: 35528402023

  Signatures: _____

Certificate

This is to certify that the project report entitled "OrderEats – Food Ordering Android App" has been carried out by Rajender Mohan Verma (BCA), Shekhar Maurya (BCA), and Anushka Singh (BCA) under my supervision. The work presented herein is satisfactory and is approved for submission in partial fulfillment of the requirements of the Bachelor of Computer Applications (BCA) as a part of the curriculum bearing Course Code - 331
In Guru Gobind Singh Indraprastha University, New Delhi – 110078.

(Project Guide Signature)
Ms. Charanpreet Kaur

Date –
Place –

CONTENTS

4. Synopsis of the Project (Executive Summary)

4.1 Introduction:

In today's fast-paced digital world, technology has transformed almost every aspect of daily life, including the way people order food. Traditionally, customers relied on manual processes like visiting restaurants in person, waiting in queues, or calling over the phone to place an order. These methods often led to errors, delays, and customer dissatisfaction. With the rapid growth of smartphones and the increasing affordability of internet access, mobile applications have become the most convenient medium for connecting customers with restaurants and delivery partners.

The project **"OrderEats"** is an Android-based mobile food ordering application developed to simplify, digitalize, and modernize the food ordering process. The aim of this project is to build an easy-to-use, student-friendly app that demonstrates the fundamental features of online food ordering systems, such as user authentication, menu browsing, cart management, order history, and user profiles.

OrderEats is structured with clean and modular architecture using Activities and Fragments for navigation, RecyclerView adapters for dynamic lists like menus and popular items, and modern UI/UX principles for smooth interaction. The project focuses on learning and training, while also solving real-world problems like miscommunication in orders, time wasted in queues, and lack of transparency in traditional methods.

The app is lightweight yet scalable. It is not only designed for training purposes but also has the potential to evolve into a fully functional commercial application. With further enhancements like payment gateway integration, GPS-based tracking, AI-powered recommendations, and a restaurant admin panel, the app could compete with popular platforms such as Zomato, Swiggy, or UberEats on a smaller scale.

This chapter provides an overview of the project's problem definition, reasons for selecting the topic, objectives, scope, and domain. Each section highlights the importance of OrderEats as a learning project and its relevance in the present-day digital ecosystem.

4.2 Statement about the Problem:

Food ordering is a daily necessity for students, working professionals, and families. However, the process has traditionally been filled with inefficiencies. Customers often face challenges such as:

1. **Human Errors** – Miscommunication over phone calls often leads to incorrect or misplaced orders.
2. **Delays in Service** – Long queues during peak hours result in wasted time and frustration.
3. **Lack of Transparency** – Customers have no visibility of order status, preparation time, or delivery timelines.
4. **Inconvenient Payments** – Manual billing and cash handling lead to delays and errors.
5. **Limited Options** – Without digital menus, customers are unable to explore available food varieties or promotions.

From the restaurant's perspective, manual order management also creates bottlenecks:

- Difficulty in handling multiple orders at once.
- Increased chance of losing customers due to long wait times.
- Inability to analyze sales data and track performance.

The core problem is the absence of a digital platform that connects users and restaurants in a structured, error-free, and time-saving manner.

OrderEats addresses this problem by offering a mobile-first solution. With just a few taps, users can log in, browse restaurant menus, select their favorite items, add them to the cart, and place orders seamlessly. Restaurants benefit from organized order management, while customers enjoy transparency, accuracy, and efficiency.

Thus, the project bridges the gap between traditional ordering inefficiencies and the modern need for speed, accuracy, and convenience.

4.3 Why is the particular topic chosen?

The decision to choose OrderEats as the project topic was guided by both relevance and practical learning opportunities.

1. **Relevance in Today's World**

   Online food ordering systems are no longer luxuries; they are necessities. Urban lifestyles, busy schedules, and the demand for convenience have made food apps extremely popular. The COVID-19 pandemic further accelerated the adoption of digital ordering, making this topic highly practical and timely.

2. **Learning Value for Students**

   By working on OrderEats, students get real-life exposure to:

   - Android app development using Kotlin/XML.
   - Database integration using Firebase.
   - User authentication and secure login systems.
   - UI/UX design principles with smooth animations.
   - System analysis, design diagrams (DFD, ER), and project planning (PERT chart).

3. **Teamwork and Collaboration**

   The project provides an opportunity to collaborate in a team, dividing responsibilities such as UI/UX design, backend/database setup, and coding. This mirrors the structure of real software development teams.

4. **Future Career Benefits**

   The project strengthens technical portfolios, giving students practical experience that can be showcased in interviews for internships and jobs.

5. **Scope for Innovation**

   Unlike static academic projects, OrderEats has real potential to grow. Future additions like AI-powered recommendations, GPS integration, and admin dashboards make it a scalable project with real-world application.

In short, the topic was chosen because it is practical, in-demand, and offers wide learning exposure in mobile app development.

4.4 Objective:

The primary objectives of the OrderEats project are divided into technical and non-technical goals:

Technical Objectives:

- User-Friendly App: To design an intuitive Android application that makes food ordering seamless for users.
- Core Features: Implement login/signup, location selection, menu browsing, cart management, and order history.
- Efficient Architecture: Adopt modular design with Activities, Fragments, RecyclerViews, and clean separation of concerns.
- Database Integration: Enable persistent data storage for users, orders, and menus using Firebase.
- Security & Validation: Implement secure authentication and validation to protect user information.
- Scalability: Design the app so that additional features like payments and GPS can be added in the future.

Non-Technical Objectives:

- Teamwork & Collaboration: Strengthen skills in communication, coordination, and role distribution.
- Documentation: Create a detailed project report covering all stages of the software development life cycle (SDLC).
- Project Management: Learn how to plan and execute tasks within deadlines using tools like PERT charts.
- Presentation Skills: Gain confidence in presenting technical projects with clarity and professionalism.

By achieving these objectives, the OrderEats project becomes not only a technical solution but also a training platform for building software engineering skills.

4.5 Scope:

The scope of OrderEats is divided into current scope and future scope:

Current Scope:

- User authentication (Login/Signup).
- Location selection for food availability.
- Browse menus and popular items.
- Add items to the cart and manage cart contents.
- View past orders in history.
- Manage user profile (name, email, location, logout).

This makes the app suitable for college students, small restaurants, and demo food outlets where manual ordering is inefficient.

Future Scope:

- Payment Gateway Integration: Secure online transactions through UPI, credit/debit cards, or wallets.
- AI-Based Recommendations: Suggest meals based on user history and preferences.
- Real-Time GPS Tracking: Track delivery partners live on the map.
- Restaurant Admin Panel: Enable restaurants to manage menus, orders, and analytics.
- Promotions & Discounts: Add features for coupons, special offers, and loyalty rewards.
- Scalability to City/National Level: Expand from local outlets to large-scale implementation.

Thus, the scope ensures that while the app is simple enough for students to build, it is powerful enough to expand into a full-fledged commercial product.

4.6 Domain:

The OrderEats project belongs to the domain of Android Application Development, with integrations from Database Management and UI/UX Design.

Key aspects of the domain include:

1. **Android Development**
    - Core principles: Activity lifecycle, Intent handling, UI rendering.
    - Use of Fragments for modular and reusable UI components.
    - Navigation components for smooth flow between app screens.

2. **Database Management**
    - Firebase to store user profiles, menus, and orders.
    - CRUD operations (Create, Read, Update, Delete) for data.
    - Data security and validation.
3. **UI/UX Design**
    - Clean and engaging interfaces with XML layouts.
    - Animations (fade, slide) to improve user experience.
    - Custom shapes, gradients, and styles for branding.
4. **Cloud Integration**
    - Use of Firebase for real-time database and authentication.
    - Scalability through cloud services.
5. **Software Engineering Principles**
    - System Analysis & Design through diagrams (DFD, ER, UML).
    - Structured documentation and project planning (PERT charts).
    - Testing methodologies to ensure reliability and usability.

By working in this domain, students gain comprehensive exposure to mobile app development, system design, and teamwork—making OrderEats an ideal project for training and academic evaluation.

Chapter 1: Objective & Scope of the Project

1.1 Introduction

In the fast-evolving landscape of digital services, mobile technology has fundamentally transformed the way consumers interact with businesses. Food, being one of the most essential human needs, is now seamlessly integrated into this digital revolution. Traditionally, customers had to either physically visit restaurants, wait in long queues, or place orders via phone calls. These methods often resulted in inefficiency, communication errors, delays, and an overall frustrating experience. With the penetration of smartphones, high-speed internet connectivity, and the rising popularity of Android applications, food ordering apps have emerged as a solution that bridges convenience, efficiency, and user satisfaction.

OrderEats is an Android-based food ordering application designed primarily as a training and learning project for students. While many commercial apps like Swiggy, Zomato, and UberEats exist in the market, the aim of OrderEats is not only to replicate the food ordering process but also to serve as a structured platform for students to practice Android application development, software design, teamwork, and project documentation. Thus, it provides dual benefits—an easy-to-use app for users and a complete hands-on learning experience for developers.

The project emphasizes clean UI design, modular code structure, and smooth navigation, enabling students to adopt good engineering practices. The overall structure of the app is built around standard Android components, where Activities manage top-level workflows (SplashActivity, StartActivity, AuthenticationActivity, LocationActivity, and MainActivity), while Fragments handle bottom navigation screens (Home, Search, Cart, History, Profile).

The lists of food items, categories, and past orders are rendered using RecyclerView with Adapters such as PopularAdapter, MenuAdapter, and CartAdapter, ensuring efficient scrolling and data rendering. Moreover, lightweight animations like fade-in, slide-in, and transition effects have been added to enhance user experience. The data model is designed to connect users, restaurants, menu items, and orders logically and can be extended in future versions.

This introduction highlights that OrderEats is more than just an app prototype. It is a well-structured academic project that teaches practical skills in Android development, problem-solving, teamwork, and software documentation.

In short, the introduction of OrderEats highlights its dual purpose: to give a working prototype of a food ordering app for users and to provide developers (students) an opportunity to practice essential concepts of Android application development.

1.2 Objective

The objectives of the OrderEats project are broad and aligned with both user needs and student learning outcomes. Below are the detailed objectives:

1.2.1 Develop a Student-Friendly Food Ordering App

The primary goal is to design and implement a food ordering application that provides a complete user journey, from logging in to placing orders. This includes Login/Signup, menu browsing, cart management, and order history tracking. By creating this app, students simulate a real-world scenario of how food ordering platforms work.

1.2.2 Practice Modular UI Composition

The project allows developers to explore modular UI development. Activities manage major screens, while Fragments are used for reusable components under navigation tabs. RecyclerView with adapters ensures list efficiency, providing students practical exposure to modern UI design patterns.

1.2.3 Implement Authentication and Data Persistence

Authentication is a crucial feature in any app. In OrderEats, secure login and signup mechanisms are implemented using Firebase Authentication or custom APIs. Basic data persistence using Firebase allows users to maintain cart data, order history, and profiles. Input validation ensures reliability.

1.2.4 Maintain Clean Architecture

The project enforces separation of concerns between UI, data, and business logic. This prepares students to write maintainable and reusable code that can scale when advanced features are added later.

1.2.5 Enhance Teamwork and Collaboration

OrderEats is designed as a team project. Tasks are divided into specific roles such as UI/UX design, backend development, business logic implementation, and documentation. This division not only improves efficiency but also simulates professional teamwork scenarios.

1.2.6 Apply Modern UI/UX Principles

Animations, gradients, custom shapes, and styles are applied to enhance user experience. Students gain practical exposure to making applications visually appealing and interactive.

1.2.7 Document the Project Life Cycle

Proper documentation is part of the project, covering phases such as requirement analysis, design, development, testing, and maintenance. This ensures the project aligns with academic standards and provides a solid reference for future improvements.

1.3 Scope:

The scope of OrderEats can be divided into two categories: Current Scope (features already implemented) and Future Scope (features that can be added later).

1.3.1 Current Scope

The current version of OrderEats includes:

- Login/Signup for user account creation and authentication.
- Menu browsing for exploring food items, categories, and offers.
- Cart management, allowing add/remove items and price calculations.
- Order history to review past orders.
- Profile settings to manage personal information.

This ensures that the app provides a complete end-to-end workflow of food ordering, even if advanced features like payments and live tracking are not included.

1.3.2 Future Scope

In the future, the app can be extended with:

- Payment integration through UPI, wallets, and cards.
- Location selection & GPS tracking for real-time deliveries.
- Restaurant dashboards for order management.
- AI-based recommendations for personalized suggestions.
- Scalability to city-level or nationwide services**.**

1.4 Relevance of the Project

Food ordering apps are among the most widely used mobile services globally. The demand for such platforms skyrocketed during the COVID-19 pandemic, where digital-first solutions became a necessity. OrderEats, though a training project, prepares students to contribute meaningfully to this growing industry.

1.5 Challenges Addressed

The project addresses several challenges faced in traditional food ordering:

- Miscommunication and errors in manual orders.
- Long waiting times in queues.
- No centralized system for tracking orders.
- Lack of convenience for customers and scalability for restaurants.

1.6 Contribution to Learning

For students, this project provides:

- Practical Android Development Skills (Activities, Fragments, RecyclerView).
- Database Management (Firebase/MySQL).
- UI/UX Design Practice with animations and styles.
- Teamwork and Documentation Skills, essential for industry readiness.

Chapter 2: Theoretical Background & Problem Definition

2.1 Theoretical Background

The "OrderEats" application is developed following the client–server paradigm, a model that forms the backbone of almost all modern mobile applications. In this paradigm, the client (Android app on the user's phone) handles user interaction, interface design, and local processing, while the **server** (Firebase/MySQL backend) manages authentication, data storage, and order processing. This division allows the system to remain scalable, maintainable, and efficient.

2.1.1 Android Client-Side Components:

On the client side, OrderEats leverages fundamental Android concepts to provide a robust and engaging user experience.

Activities and Fragments

- **Activities** act as entry points for user workflows. Each major part of the app — Splash Screen, Login/Signup, Location selection, or the Main Dashboard — is represented by an Activity.
- **Fragments** divide the UI into reusable sections like Home, Search, Cart, History, and Profile. By embedding multiple fragments within one Activity, the app achieves modularity.
- Example: The CartFragment can be reused in both the Cart workflow and during the Checkout process.

This modular design is important because it promotes code reusability, reduces duplication, and allows developers to maintain or extend features more easily. For students building apps, it provides real exposure to real-world software architecture.

2.1.2 RecyclerView and Adapters

The RecyclerView widget is one of the most powerful UI tools in Android. Unlike traditional layouts, which slow down with large data sets, RecyclerView is optimized to **recycle** views and reuse memory efficiently.

- **Adapters** bridge the gap between the dataset and UI.
    - **MenuAdapter** → Displays menu items with pictures, prices, and add-to-cart buttons.
    - **CartAdapter** → Shows the user's current selection and allows quantity updates.
    - **PopularAdapter** → Highlights trending or promotional items.

This design ensures smooth scrolling, even when the menu contains dozens or hundreds of items.

Comparison: A ListView was the older solution but lacked flexibility. RecyclerView allows complex designs (images, buttons, nested layouts) and animations, making it the industry standard.

2.1.3 Navigation Component

Navigation in Android can get complicated when managing multiple fragments and activities. OrderEats uses the Navigation Component to simplify this:

- Provides a clear navigation graph.
- Manages the back stack automatically.
- Reduces boilerplate code.

For example, when a user logs in, they are directly routed to the Main Dashboard, and the back button does not return them to the Login page — thanks to structured navigation handling.

2.1.4 ViewModel and Lifecycle Management

A key challenge in Android is configuration changes like screen rotation. Without proper handling, the app can lose data (e.g., items in the cart disappearing).

- ViewModel stores UI-related data so that it survives configuration changes.
- Lifecycle Awareness prevents memory leaks by ensuring components run only when needed.

This makes OrderEats more reliable, especially when students test it on different devices.

2.1.5 Animations and Transitions

Modern users expect apps to "feel alive." OrderEats uses animations such as:

- **Fade-in** → Smooth entry of menu items.
- **Slide transitions** → Between Cart and Checkout pages.
- **Button animations** → Feedback when items are added.

These animations not only improve aesthetics but also reduce perceived wait time, making the app more engaging.

Shapes, Styles, and Design Tokens

Design consistency is vital. OrderEats employs:

- XML drawables (e.g., *greenButtonGradient.xml*) for gradient buttons.
- Centralized style files for fonts, padding, and colors.
- Reusable components so changes can be applied globally.

This ensures a professional look and makes updates easier.

2.1.6 Server-Side Components

The server side ensures persistent storage, authentication, and synchronization. OrderEats is designed to work with Firebase (cloud-based, real-time database).

- Firebase provides easy authentication, cloud storage, and push notifications — ideal for prototypes and small-scale apps.

In this project, Firebase can handle user authentication (email/password login) and menu storage.

2.1.7 Data Modeling

Data is structured to reflect real-world relationships:

- **User** → Can place multiple Orders.
- **Order** → Contains one or more MenuItems.
- **Cart** → Acts as a temporary container before an Order is finalized.

This entity relationship ensures clarity and extensibility. For future versions, entities like Restaurant, Delivery Partner, and Payments can be added without major redesigns.

2.1.8 Security and Validation

Security is a crucial part of mobile applications. OrderEats addresses it through:

- **Input Validation** → Prevents invalid data from crashing the app.
- **Password Hashing** → For non-Firebase systems, ensures credentials are encrypted.

- **Access Rules** → Firebase Database rules restrict unauthorized data access.
- **Minimal PII** → Only necessary user data (name, email, address) is stored.

By keeping security simple yet effective, the app balances ease of development with reliability.

2.1.9 Why Modular & Scalable Design Matters

OrderEats emphasizes scalability so students can extend it later:

- Adding Payment Gateway → UPI/Wallet integration.
- Adding GPS Tracking → For real-time order delivery.
- Extending to Multi-Restaurant Support.

Thus, even though the current version is basic, the theoretical foundation ensures it can grow into a full-scale commercial system.

2.2 Problem Statement

Despite advancements in technology, food ordering still faces challenges in many areas.

2.2.1 Traditional Ordering Issues

- Phone Calls: Prone to miscommunication.
- Physical Queues: Wasted time during peak hours.
- Manual Billing: Higher chance of human errors.
- Lack of Tracking: Customers remain uncertain about order status.

These limitations frustrate customers and reduce satisfaction.

2.2.2 Customer Challenges

1. Difficulty in exploring menus without visuals.
2. No transparency in order progress.
3. Inconvenience of carrying cash or waiting for bills.
4. Frustration due to delays and errors.

2.2.3 Restaurant Challenges

1. Hard to manage peak-hour demand.
2. No structured system for recording orders.
3. Missed revenue opportunities due to inefficiency.
4. Difficulty in managing customer loyalty.

2.2.4 Need for a Mobile-First Solution

Apps like Zomato and Swiggy show how powerful digital food ordering can be, but they are large-scale systems. Students can replicate core features through a training app like OrderEats:

- Simple Login/Signup.
- Clear Menus with Pictures & Prices.
- Cart System for Review.
- Order History for Transparency.

By focusing on these essentials, OrderEats solves the gap between traditional ordering and full-scale commercial apps.

2.2.5 Why OrderEats is an Ideal Student Project

- Practical exposure to Android Development.
- Application of Database Management.
- Hands-on learning of UI/UX principles.
- Opportunities for teamwork and project management.
- Foundation for adding advanced features later.

OrderEats thus serves both as a solution for food ordering inefficiencies and as a learning platform for students.

Chapter 3: System Analysis & Design vis-à-vis User Requirements

3.1 System Analysis

System analysis is one of the most important phases of the Software Development Life Cycle (SDLC). In this phase, developers and analysts closely study the problem domain, identify the requirements of the stakeholders, and define how the software system should function. For the OrderEats project, system analysis focuses on understanding the needs of customers, restaurant owners, and administrators, and translating them into clear functional and non-functional requirements. A strong analysis ensures that the app will be user-friendly, efficient, scalable, and maintainable.

3.1.1 Understanding User Roles

Identifying user roles is the first step in system analysis. In the current training version of OrderEats, the primary user is the Customer, but in the future, additional roles can be introduced.

1. **Customer (End User)**
    - The main user of the app.
    - Can browse menus, add items to the cart, place orders, and view order history.
    - Needs simple navigation, clear menus, and transparent pricing.
    - Example: A student sitting in the college canteen wants to quickly browse available dishes and place an order without standing in long queues.
2. **Admin** *(future extension)*
    - Manages the overall system, users, and data.
    - Responsible for adding or removing restaurants, approving menu updates, and monitoring app usage.
    - Can generate reports on sales, order frequency, and system activity.
3. **Restaurant Owner** *(future extension)*
    - Manages their restaurant profile, menu, pricing, and availability.
    - Receives notifications of incoming orders, accepts/rejects orders, and updates preparation status.
4. **Delivery Partner** *(future extension)*
    - Assigned to deliver confirmed orders.

- Can view pickup and delivery addresses on a map, track delivery time, and update order completion status.

By clearly defining roles, the system is future-proof, meaning it can easily scale without redesigning the entire architecture.

3.1.2 Functional Requirements

Functional requirements describe what the system should do. For OrderEats, these are the key features:

1. **Account Management**
   - New users can sign up using email, phone number, or social login.
   - Existing users can log in securely with authentication checks.
   - Users can edit their profiles (name, email, phone number, saved addresses).
   - Example: A user changes their delivery address from hostel to classroom during signup.
2. **Menu Browsing**
   - Users can view items categorized by type (Beverages, Snacks, Meals, etc.).
   - Popular and recommended items are highlighted.
   - Menus include images, prices, and short descriptions.
3. **Cart Management**
   - Items can be added, removed, or updated in quantity.
   - Real-time total price calculation is shown.
   - Cart acts as a "temporary storage" before final order placement.
4. **Order Placement & Processing**
   - Orders move through statuses like *Placed → In Preparation → Ready → Delivered*.
   - Users receive notifications (in-app or push) about updates.
   - Mocked order tracking is available for the training version.
5. **Order History**
   - Users can view a list of all previous orders with details.
   - Helps repeat favorite orders with a single click.

6. **Search & Filters**
    - Users can search by item name or filter by category, price, or offers.

3.1.3 Non-Functional Requirements

Non-functional requirements describe how the system should behave in terms of quality.

1. **Usability**
    - Simple navigation, clear buttons, and minimal steps for placing an order.
    - Example: A new user should be able to place their first order within 2–3 minutes.
2. **Performance**
    - The app should load menus within 2–3 seconds even with large datasets.
    - RecyclerView ensures smooth scrolling of long menus.
3. **Reliability**
    - The app should handle network failures gracefully.
    - If internet is lost during checkout, the order should be saved temporarily.
4. **Maintainability**
    - Code should be modular (Activities, Fragments, Adapters).
    - Developers can easily add new features without rewriting existing code.
5. **Scalability**
    - The system must support thousands of users in the future.
    - Database design should allow easy integration with new features like payments, AI recommendations, or delivery tracking.
6. **Security**
    - Input validation for all fields (e.g., no invalid email or empty password).
    - Authentication via Firebase or encrypted passwords.
    - Database access restricted with security rules.

3.1.4 Summary of System Analysis

The analysis phase ensures that the app meets both user needs and technical requirements. With well-defined roles, functional and non-functional requirements, the foundation is set for a robust and future-ready system.

3.2 System Design:

System design translates the requirements gathered in the analysis phase into a structured architecture. It defines how modules will interact, how data will flow, and how the UI will be organized. The aim is to make the system modular, reusable, and scalable, while ensuring an engaging user experience.

3.2.1 High-Level Modules

1. **Activities**
    - **SplashScreenActivity** – Shows app logo and initializes resources.
    - **StartActivity** – Provides options for login/signup.
    - **LoginActivity / SignUpActivity** – Handles authentication.
    - **ChooseLocationActivity** – Lets users select delivery location.
    - **MainActivity** – The central hub hosting fragments.
2. **Fragments**
    - **HomeFragment** – Shows offers, featured items, categories.
    - **SearchFragment** – Enables keyword-based search.
    - **CartFragment** – Displays selected items and checkout options.
    - **HistoryFragment** – Shows previous orders.
    - **ProfileFragment** – Allows editing user details.
    - **MenuBottomSheetFragment** – Popup for quick menu customization.
3. **Adapters**
    - **PopularAdapter** – Displays trending dishes.
    - **MenuAdapter** – Handles the full menu list.
    - **CartAdapter** – Manages cart item rendering and updates.
4. **UI Assets**
    - Buttons: whiteButton.xml, proceedButtonShape.xml, greenButtonGradient.xml.
    - Text fields and shapes for consistent branding.
5. **Navigation**
    - Implemented using navigation.xml for defining flow between fragments.
6. **Animations**
    - fade_in.xml, slide_in_from_left.xml, slide_in_from_right.xml.
    - Smooth transitions to improve user engagement.

3.2.2 Data Design

The database is designed with relationships:

- **User** (UserID, Name, Email, Phone, Password)
- **Order** (OrderID, UserID, Date, Status, TotalPrice)
- **MenuItem** (ItemID, Name, Category, Price, ImageURL)
- **Cart** (CartID, UserID, ItemID, Quantity)

Future entities: Restaurant, Delivery Partner, Payment.

3.2.3 Security Design

- Authentication using Firebase Auth (email/password, Google sign-in).
- Database rules restrict access only to authenticated users.
- Passwords hashed if stored in MySQL.
- No sensitive financial data stored in training version.

3.2.4 Design Rationale

- Activities manage entry points.
- Fragments provide reusability.
- Adapters ensure performance with lists.
- XML assets centralize UI design.
- Modular design makes scaling and testing easier.

3.2.5 Future Design Extensions

- Payment gateway integration.
- Restaurant dashboards.
- AI-powered recommendations.
- Real-time delivery tracking with GPS.
- Multi-language support for wider accessibility.

3.3 Conclusion

System Analysis and Design together create a blueprint for the OrderEats app. Analysis ensures that the system aligns with user needs and business objectives, while design converts those requirements into a modular, scalable, and secure architecture. The result is a training project that not only delivers a working food ordering application but also provides valuable learning in Android development, database design, and UI/UX principles.

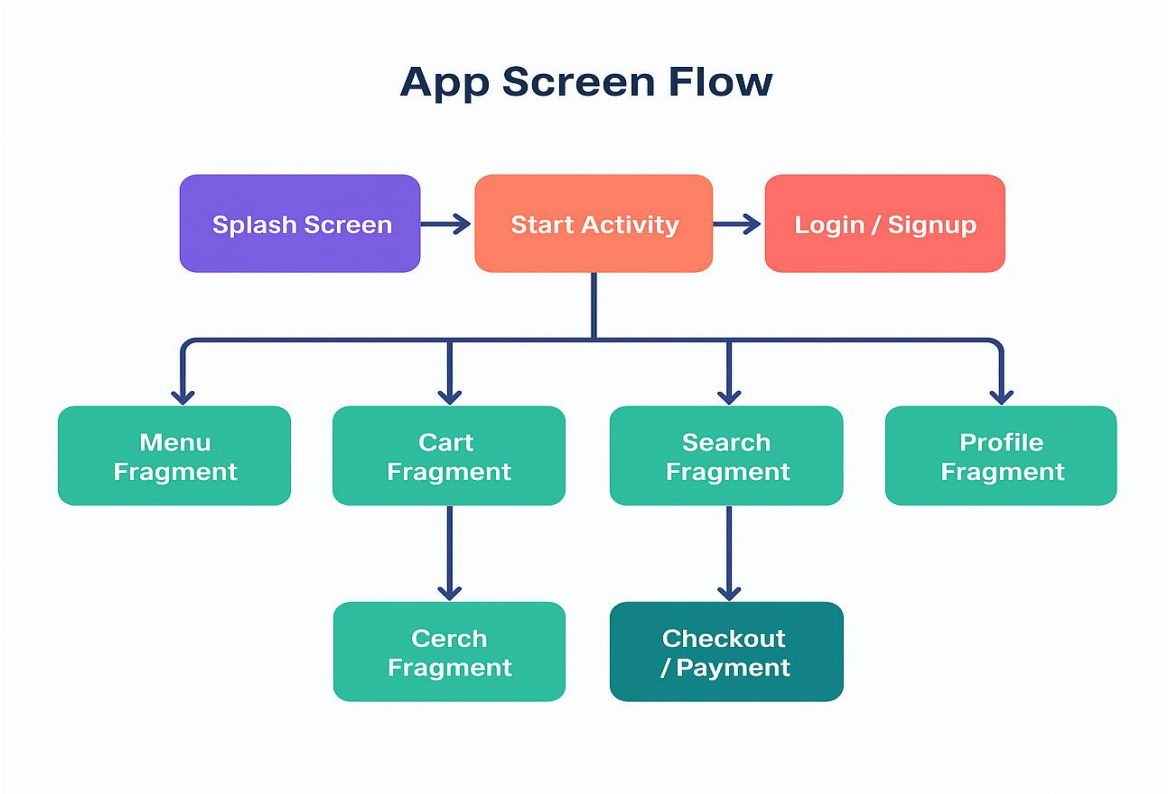


Figure I: App Screen Flow

Chapter 4: System Planning (PERT Chart)

4.1 Introduction to System Planning

System planning is one of the most important phases of any software development life cycle. Without a clear plan, even the best ideas and the most talented teams can face delays, resource mismanagement, and unexpected project failures. Planning ensures that tasks are not only identified but also scheduled, sequenced, and tracked effectively. It is a strategic step that connects theoretical requirements and design with practical implementation and deployment.

In the context of the OrderEats project, system planning plays a crucial role because the project is being carried out in a limited timeframe (6 weeks) as a training-oriented student project. The aim is not just to deliver a working food ordering app but also to gain hands-on experience in applying project management principles, such as scheduling, resource allocation, and dependency tracking.

For this project, a PERT (Program Evaluation and Review Technique) chart is adopted as the main planning tool. The PERT chart helps visualize tasks, their order, and the dependencies among them, which is extremely valuable when multiple phases such as requirement gathering, UI/UX design, database setup, coding, testing, and deployment need to be completed systematically within six weeks.

4.2 What is a PERT Chart?

A PERT chart is a project management tool developed in the 1950s to manage complex defense projects in the United States. Today, it is widely used in software development and engineering projects. It is essentially a flowchart that graphically represents project tasks, their durations, and dependencies.

The core components of a PERT chart are:

- **Tasks (or Activities):** Represented by nodes or circles, showing specific work units.
- **Dependencies:** Arrows connecting tasks to indicate order.
- **Critical Path:** The longest path of dependent tasks that determines the minimum project duration.

For the OrderEats project, the PERT chart helps ensure that no phase is skipped, deadlines are realistic, and critical tasks (like coding and testing) are carefully tracked.

4.3 Importance of System Planning in OrderEats

Before diving into the 6-week breakdown, it is important to highlight why system planning is essential:

1. **Clarity of Objectives:** Planning helps translate abstract objectives (like "create a food ordering app") into measurable milestones.
2. **Time Management:** With a fixed six-week timeline, each week must be optimally utilized to ensure timely completion.
3. **Dependency Tracking:** Some tasks cannot start until others are finished (for example, implementation cannot begin without a database setup). The PERT chart makes these dependencies explicit.
4. **Resource Utilization:** Team members can be assigned specific modules without overlap or conflict.
5. **Risk Minimization:** Early identification of potential bottlenecks allows contingency plans.

4.4 Project Timeline (6 Weeks)

Unlike typical semester projects spanning 8–10 weeks, the OrderEats project is condensed into 6 weeks for training purposes. Each week has clearly defined deliverables:

Week 1 – Requirement Gathering and Analysis

The first week is dedicated to gathering and analyzing requirements from both technical and user perspectives.

Activities:

- Interacting with mentors and teammates to finalize scope.
- Identifying functional requirements (login, signup, browsing menus, cart, history).
- Identifying non-functional requirements (usability, performance, scalability).
- Documenting requirements in an SRS (Software Requirements Specification).

Deliverables:

- SRS Document
- Defined user roles (Customer, Admin, future expansion: Restaurant Owner)

Week 2 – UI/UX Design Phase

Once requirements are clear, the second week is used to create the visual design and navigation structure.

Activities:

- Designing wireframes for all major screens: Splash, Login, Main Dashboard, Cart, Profile, and History.
- Preparing a navigation flow diagram showing how users move across screens.
- Choosing color themes, typography, and branding consistent with modern design standards.
- Prototyping using tools like Figma or Adobe XD for stakeholder feedback.

Deliverables:

- High-fidelity mockups for each screen
- Navigation flow document
- Design tokens (color palette, gradients, styles)

Week 3 – Database Setup

The third week focuses on preparing the backend infrastructure.

Activities:

- Designing an ER diagram showing entities like User, Menu, Orders, Cart.
- Creating relational tables in Firebase/MySQL.
- Defining constraints, keys, and relationships (e.g., one User → many Orders).
- Populating tables with sample data for testing.
- Testing queries to ensure data retrieval and storage are correct.

Deliverables:

- Fully functional database schema
- Connected database ready for integration.

Week 4 – Implementation Phase

This is the most time-intensive week, as actual coding and integration are performed.

Activities:

- Developing Activities: SplashActivity, StartActivity, AuthenticationActivity, MainActivity.

- Coding Fragments: Home, Search, Cart, Profile, History.
- Creating Adapters: MenuAdapter, CartAdapter, PopularAdapter.
- Integrating Navigation Component for smooth back-stack handling.
- Implementing UI theming, animations (fade, slide-in).

Deliverables:

- Working prototype with functional UI and backend connections
- Source code in structured packages (activities, fragments, adapters, models, utils)

Week 5 – Testing Phase

The fifth week is dedicated to validating functionality and fixing errors.

Activities:

- **Unit Testing:** Ensuring each module works independently.
- **Integration Testing:** Checking if modules interact correctly (e.g., Cart and Orders).
- **Usability Testing:** Ensuring UI is intuitive.
- **Performance Testing:** Measuring response time and memory efficiency.
- **Bug Fixing:** Resolving crashes, navigation issues, or incorrect data handling.

Deliverables:

- Bug-free system
- Testing report documenting all test cases and outcomes

Week 6 – Deployment and Documentation

The final week focuses on delivering a polished product.

Activities:

- Final integration with Firebase for authentication and storage.
- Building APK/AAB release versions.
- Deploying to Google Play (optional for training).
- Preparing final documentation (SRS, Design, Testing, User Manual).
- Conducting project presentation.

Deliverables:

- Fully functional deployed version
- Complete documentation bundle
- Final presentation slides

4.5 PERT Chart for 6-Week Timeline

The PERT chart graphically represents the tasks above. The chart highlights sequential dependencies, for example:

- Requirement gathering (Week 1) must be completed before design (Week 2).
- Database setup (Week 3) must be finished before implementation (Week 4).
- Implementation must be completed before testing (Week 5).

The critical path is:

Requirement → UI/UX Design → Database Setup → Implementation → Testing → Deployment

Any delay in these stages directly delays the entire project.

4.5.1 Advantages of PERT Chart:

1. Clear Visualization: Provides a clear graphical representation of tasks, their sequence, and dependencies.
2. Identifies Critical Path: Helps focus on tasks that directly affect project completion time.
3. Efficient Resource Allocation: Assists in planning and allocating resources effectively.
4. Time Management: Helps estimate project duration and manage deadlines better.
5. Risk Reduction: Early identification of potential delays allows corrective actions to be taken.

4.5.2 Disadvantages of PERT Chart:

1. Complex for Large Projects: Can become complicated and hard to manage for very large projects with many tasks.
2. Estimation Errors: Relies on accurate time estimates; wrong estimates can affect the entire schedule.
3. Time-Consuming: Preparing and updating the chart can take considerable time and effort.
4. Does Not Guarantee Success: Even with a PERT chart, unforeseen issues may still delay the project.

4.8 Challenges in Planning OrderEats

- Limited six-week timeline
- Balancing parallel tasks (UI design vs. backend setup)
- Ensuring synchronization among team members
- Estimating accurate task durations without prior experience

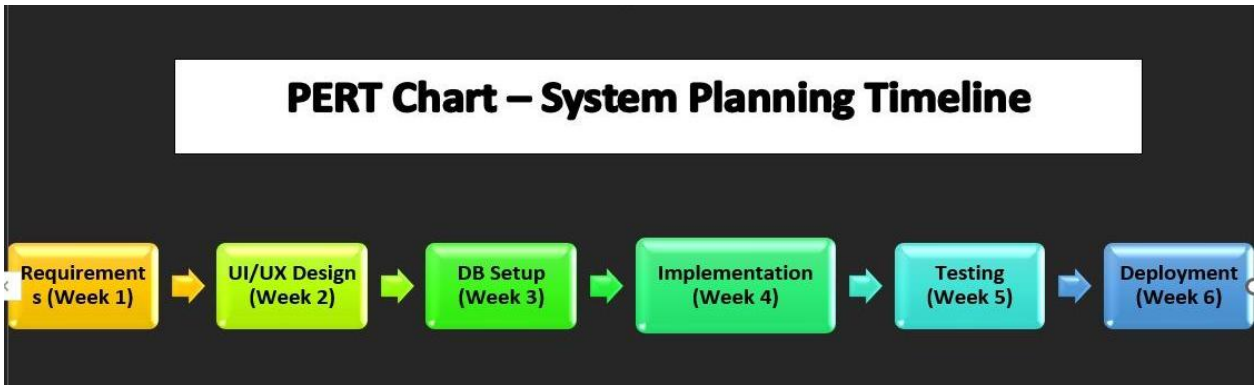


Figure II: PERT Chart – System Planning Timeline

Chapter 5: Methodology, Implementation, Hardware & Software, Maintenance & Evaluation

5.1 Methodology Adopted

Software development methodology is the foundation for how the project is structured, executed, and delivered. In this project, we adopted an Agile-inspired, iterative methodology, which was particularly well-suited for a student team environment where flexibility and adaptability are essential. Unlike the Waterfall model, which emphasizes a linear and sequential flow, Agile allows dynamic changes and feedback incorporation at every stage.

5.1.1 Agile vs. Waterfall:

Waterfall Approach: Involves fixed stages such as requirement gathering, design, coding, testing, and deployment, with little room for modification once a stage is completed. This is useful for projects with rigid requirements but risky for projects with evolving needs.

Agile Approach: Breaks down the project into sprints (short cycles of development). At the end of each sprint, the team demonstrates a working module, gathers feedback, and improves upon it. Agile thrives in environments where requirements may shift during development.

In our context, Agile worked better because:

Requirements often evolved as we experimented with design.

Feedback from peers and faculty was incorporated quickly.

Each sprint ended with tangible progress, which was motivating for the team.

5.1.2 Sprint-Based Development

We divided the project into five sprints, each lasting around one to two weeks:

Sprint 1 – Authentication & Navigation: Focused on creating signup, login, and logout functionality with Firebase Authentication. Implemented navigation between activities like Splash, Login, and Main.

Sprint 2 – Menu & Popular Lists: Added browsing features, implemented RecyclerView for popular items, and linked categories with the full menu.

Sprint 3 – Cart & Checkout: Integrated cart functionality with live total updates, quantity adjustments, and checkout flow.

Sprint 4 – History & Profile: Built user profile editing (name, contact, addresses) and order history tracking.

Sprint 5 – Polish & Testing: Conducted bug fixes, design refinements, and accessibility improvements.

Each sprint ended with a demo where progress was showcased to peers and instructors. This iterative loop ensured that the system was functional at every stage.

5.1.3 Daily Stand-ups

The team also practiced informal stand-up meetings, where each member shared:

What they accomplished yesterday

What they planned for today

Any blockers encountered

This practice improved coordination and ensured no task was left unattended.

5.1.4 Team Roles and Responsibilities

UI/UX Lead (Member 1): Designed XML layouts, color schemes, gradients, button styles, and animations. Handled accessibility concerns such as font size, tap targets, and contrast.

Backend & Data Lead (Member 2): Managed Firebase Authentication, Realtime Database/Firestore, and data seeding for testing. Ensured secure user login and smooth synchronization.

Business Logic Lead (Member 3): Implemented adapters, managed cart logic, applied form validations, and handled interactions between fragments and activities.

This role division mirrored real-world software teams and encouraged collaboration and accountability.

5.2 System Implementation

System implementation translates requirements and design into working code. To maintain modularity and scalability, the system was broken down into activities, fragments, adapters, navigation components, and UI assets.

5.2.1 Activities Implemented:

SplashScreenActivity: Displayed logo and initialized Firebase services.

StartActivity: Guided new users to login or signup screens.

LoginActivity & SignActivity: Managed user authentication with Firebase.

ChooseLocationActivity: Allowed users to set GPS-based or manual location.

MainActivity: Hosted fragments such as Home, Cart, Profile, and History.

5.2.2 Fragments Implemented:

HomeFragment: Displayed featured and popular items.

MenuFragment: Displayed categories and vertical lists of menu items.

CartFragment: Handled cart management, quantity updates, and checkout.

HistoryFragment: Showed past orders with date and total.

ProfileFragment: Allowed users to update their details.

5.2.3 Adapters Implemented:

PopularAdapter: Displayed popular items in a card grid.

MenuAdapter: Rendered category-specific items.

CartAdapter: Managed cart lists, totals, and stepper controls.

5.2.4 Key Implementation Highlights

Authentication:

Firebase used for login/signup.

Input validation ensured correct data.

Optional “Remember Me” checkbox saved sessions.

Menu & Popular Items:

Menu items loaded from Firebase database.

Images, prices, and ratings displayed.

Tabs separated items by category.

Cart System:

Quantity steppers updated totals in real time.

Sticky bottom bar displayed current total.

Checkout validated user input before placing order.

Order History:

Orders stored in Firebase with timestamps.

Past orders displayed for quick reference.

Profile Management:

Editable profile fields stored securely in Firebase.

Profile synced across devices.

5.2.5 Engineering Details

RecyclerView Optimization:

Used setHasFixedSize(true) and efficient ViewHolders for smooth scrolling.

State Management:

Handled screen rotations using savedInstanceState. Optional ViewModels used for persistent data.

Error Handling:

Network errors displayed with Snackbars. Input errors shown inline.

UI Theming:

Implemented consistent button styles, accessible contrast ratios, and larger tap zones for usability.

5.2.6 Example Data Flow

User adds item to cart → Adapter updates view → Data pushed to Firebase.

Firebase updates trigger UI refresh → CartFragment recalculates totals.

On checkout → Order stored in Firebase → Order ID displayed in HistoryFragment.

This flow ensured real-time synchronization and reliability.

5.3 Details of Hardware and Software Used

5.3.1 Hardware Requirements: Laptop (8 GB RAM or more recommended), Android phone for on-device testing.

Processor: Quad-Core (minimum recommended Intel i5 or equivalent)

Storage: At least 10 GB of available space for Android Studio, Firebase SDKs, and project files.

5.3.2 Software Requirements:

Android Studio: Primary IDE for developing the Android application using Java.

Java Development Kit (JDK): Java was the primary programming language used for Drapin.

Firebase SDKs: Integrated with Android Studio for Firebase Authentication, Realtime Database/Firestore, and Analytics.

Database (Firebase Realtime Database/Firestore): Used for backend data storage and synchronization.

Adobe XD or Figma : Design tool for wireframes and mockups to plan the UI and UX.

Git/GitHub: Version control system to track changes and manage project code

5.3.3 Testing Devices:

Android Smartphone (running Android 7.0 or above) to test the app's performance and user experience on actual devices.

Emulator within Android Studio for testing on different screen sizes and Android versions.

5.3.4 Comparisons

- **Firebase:** Firebase chosen for scalability and real-time sync.server setup and APIs.
- **Emulator vs Real Device:** Emulator useful for testing multiple devices. Real device testing showed actual performance.

5.4 System Maintenance

System maintenance ensures long-term stability and security. It includes:

Types of Maintenance

Corrective Maintenance: Fixing bugs and errors detected after deployment.

Adaptive Maintenance: Updating app to work with new OS versions or devices.

Perfective Maintenance: Improving UI, adding animations, enhancing usability.

Preventive Maintenance: Regular updates to libraries and dependencies to prevent future issues.

Practices Adopted

Version Control: Git branches for feature, develop, and main.

Crash Reporting: Firebase Crashlytics tracked issues.

Library Updates: Regular updates to SDKs ensured security.

User Feedback Integration: Suggestions from testers used for refinements.

5.5 System Evaluation

Evaluation ensured that the app met its goals.

Criteria Used

1. **Usability:** New users could place orders in 3 taps per screen.
2. **Performance:** Achieved 60 fps scrolling with RecyclerView optimizations.
3. **Reliability:** Network errors handled gracefully with retry prompts.
4. **Code Quality:** Code modular, structured by packages, and commented.
5. **Cross-Device Compatibility:** Tested on multiple screen sizes using emulator.
6. **User Feedback:** Positive feedback for clean UI and cart handling; suggestions included adding payment and real-time tracking.

Future Scope

- **Payment Integration:** UPI, Razorpay, or PayPal.
- **Live Order Tracking:** Real-time delivery using Google Maps API.
- **Push Notifications:** For order confirmation, offers, and delivery updates.
- **AI Recommendations:** Suggest items based on order history.
- **Admin Panel:** For restaurant owners to manage items and offers.

Chapter 6: Detailed Life Cycle of the Project

The life cycle of a project defines the structured stages through which the system evolves, from the initial conceptualization to its deployment and continuous maintenance. In the context of the OrderEats food ordering system, the life cycle covers not only the design and implementation but also the representation of system processes, data flow, entity relationships, input-output design, and rigorous testing methodologies. This chapter provides a detailed exploration of the system life cycle, focusing on Data Flow Diagrams (DFD), Entity-Relationship (ER) diagrams, Input/Output Screen Designs, Processes Involved, and Testing.

6.1 Data Flow Diagrams (DFD)

Data Flow Diagrams are one of the most powerful tools used in structured system analysis and design. They provide a visual representation of how data moves through the system, how it is processed, and how it interacts with external entities.

The OrderEats app uses DFDs to illustrate the interaction between the customer, restaurant, delivery partner, and the backend system. Each level of the DFD progressively adds detail to provide clarity on the processes involved.

The DFDs are presented in two major levels:

- **Level 0 DFD (Context Level)** → A high-level overview of the entire system as a single process.
- **Level 1 DFD** → A detailed breakdown of the main process into multiple sub-processes.

6.1.1 Zero Level DFD (Context Diagram)

The **Level-0 DFD**, also called the context diagram, provides the highest-level abstraction of the OrderEats system. It views the entire food ordering platform as a single process.

Entities and Processes in Level 0 DFD

1. **Customer**
    - Provides: Orders for food.
    - Receives: Receipt after successful order placement.
    - Role: The primary user of the system who initiates transactions.
2. **Kitchen**
    - Receives: Food Order details.

- Role: Responsible for preparing food items as per the order.

3. **Restaurant Manager**
    - Receives: Bills and Management Reports.
    - Role: Oversees operations, sales, and inventory.
4. **Food Ordering System (Central Process)**
    - Acts as the main hub for all interactions.
    - Converts customer requests into orders for the kitchen and reports for management.

Flow of Data in Level 0

- Customer places Orders for food → Food Ordering System.
- System generates a Receipt → Customer.
- System sends Food Order → Kitchen for preparation.
- System sends Bill + Management Report → Restaurant Manager for monitoring.

Significance of Level 0 DFD

- Provides a big-picture view of the system.
- Identifies all external entities and their interactions.
- Useful for both technical and non-technical stakeholders.

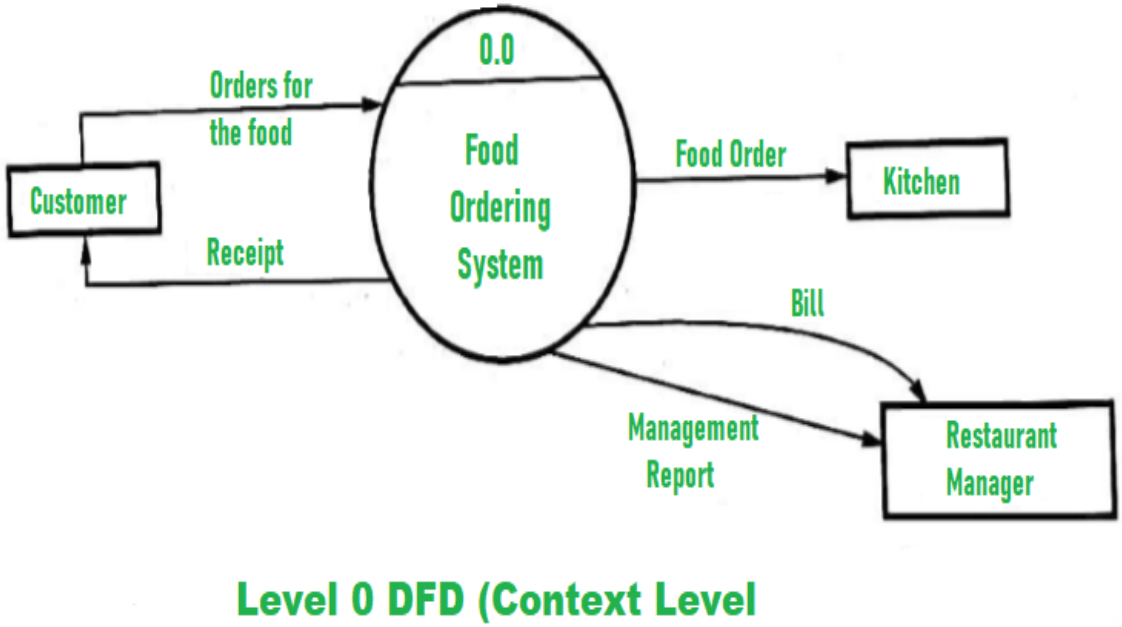


Figure III: Level – 0 DFD

6.1.2 One Level DFD

The **Level 1 DFD** decomposes the single process of Level 0 into multiple sub-processes, showing how the system internally handles customer requests.

Main Sub-processes in Level 1 DFD

1. **Processing of an Order (1.0)**
   - Input: Order from Customer.
   - Output: Food Order (to Kitchen) and Receipt (to Customer).
   - Also generates Sold Items data for further processing.
2. **Update Sold Items File (2.0)**
   - Input: Sold Items data from Process 1.0.
   - Output: Updated Sold Items File in the database.
   - Ensures sales are recorded accurately for analysis and reporting.
3. **Update Inventory File (3.0)**
   - Input: Inventory data from sold items.
   - Output: Updated Inventory Database.
   - Maintains stock availability by deducting used ingredients.
4. **Generate Management Report (4.0)**
   - Input: Data from Sold Items + Inventory depletion details.
   - Output: Management Report for the Restaurant Manager.
   - Helps in business analysis, planning, and stock control.

Entities in Level 1

1. **Customer**
   - Still interacts by placing orders and receiving receipts.
2. **Kitchen**
   - Receives food orders for preparation.
3. **Restaurant Manager**
   - Receives management reports for monitoring.
4. **Databases**
   - Database of Sold Items – Keeps track of sales.
   - Inventory Database – Stores updated stock information.

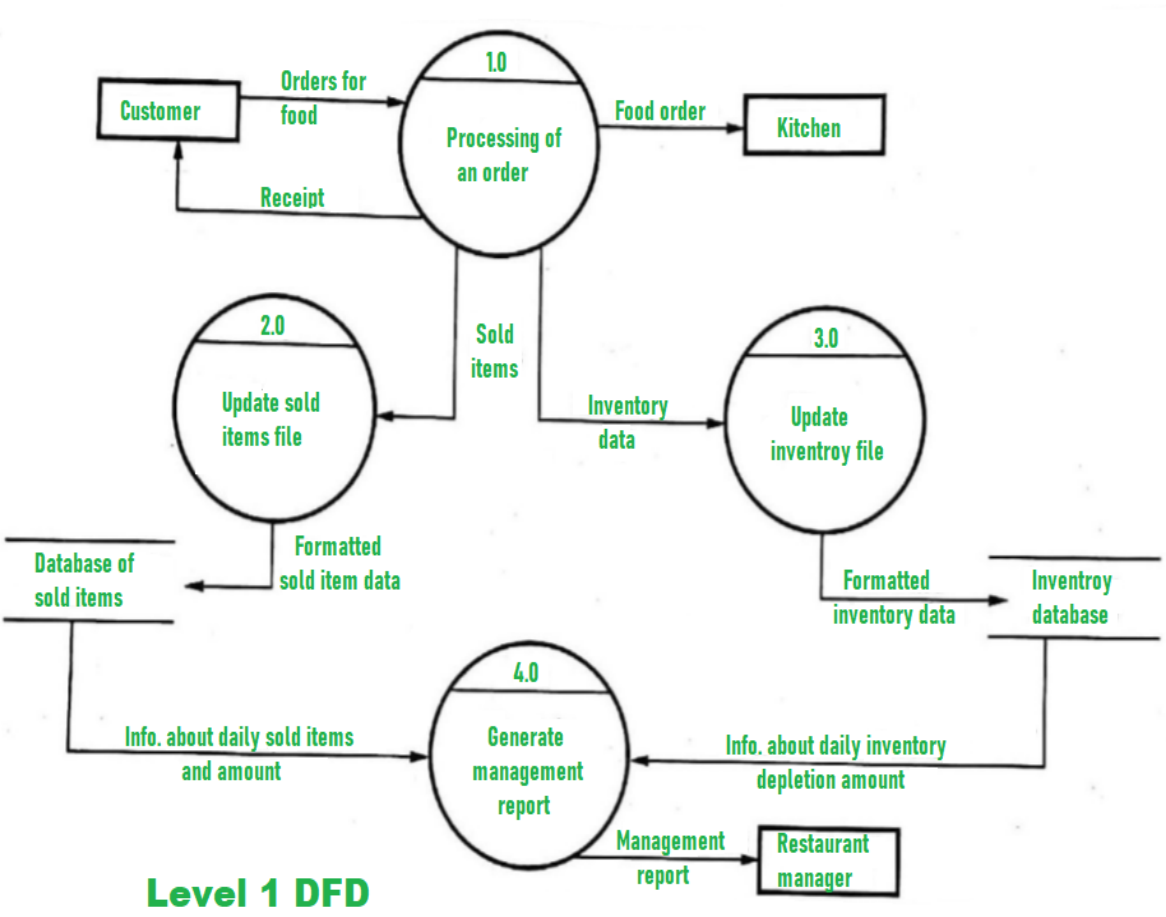


Figure IV: Level – 1 DFD

6.1.3 Level-2 DFD (Extended View)

A Level-2 DFD breaks the sub-processes further into granular details.

1. **Order Placement**
    - Customer selects food items.
    - System validates availability.
    - Cart is updated.
2. **Order Processing**
    - Calculates price and taxes.
    - Applies discounts and offers.
    - Confirms order details.
3. **Inventory Management**
    - Deducts items sold from stock.
    - Sends alerts if items run low.
4. **Billing**

- Generates invoice with payment details.
- Updates financial records.

5. **Delivery Assignment**
    - Assigns delivery person based on availability and location.
    - Sends notification to customer about estimated delivery time.

6.1.4 Advantages of DFD:

1. **Easy to Understand** – Simple diagrams that can be understood by both technical and non-technical users.
2. **Clear System Visualization** – Provides a clear overview of inputs, processes, outputs, and data stores.
3. **Improves Communication** – Enhances communication between developers, users, and stakeholders.
4. **Structured Analysis Tool** – Very useful during requirement analysis and system design phases.
5. **Error Identification** – Helps to identify missing processes, data redundancy, or incorrect flows.
6. **Stepwise Refinement** – Can be expanded from Level 0 to detailed Level n for better clarity.

6.1.5 Disadvantages of DFD:

1. **Time-Consuming** – Creating and maintaining DFDs for complex systems takes significant time.
2. **No Control Information** – Only shows data flow, not control flow or decision-making logic.
3. **Possible Ambiguity** – If proper symbols or naming conventions are not followed, it can cause confusion.
4. **Complexity in Large Systems** – Diagrams can become very complex and cluttered for large systems.
5. **No Data Storage Details** – Does not provide details on how data will be physically stored (e.g., database design).
6. **Maintenance Difficulty** – Needs frequent updates whenever system requirements change.

6.1.6 Detailed Explanation of Data Flow:

Step 1: Customer Places Order

- Customer provides order details (items, quantity, location, etc.).
- Order goes into Processing of an Order (1.0).
- Customer receives a Receipt as confirmation.

Step 2: Order Processing

- The system converts the order into two outputs:
    1. Food Order → Kitchen (for preparation).
    2. Sold Items Data → Update Sold Items File (2.0).

Step 3: Sold Items Update

- The sold item data is formatted and stored in the Database of Sold Items.
- This ensures proper record-keeping of all transactions.

Step 4: Inventory Update

- The system deducts required ingredients from stock and updates the Inventory File (3.0).
- The updated data is stored in the Inventory Database.

Step 5: Management Report

- Data from Sold Items File + Inventory Database is compiled into a Management Report (4.0).
- This report provides details such as:
    - Daily sales.
    - Inventory depletion.
    - Revenue summaries.
- The report is sent to the Restaurant Manager.

6.1.7 Limitations of DFD

1. **Time-Consuming** – Requires effort to prepare and update.
2. **No Control Flow** – Focuses on data only, not decision-making logic.
3. **Complexity in Large Systems** – Can become difficult to manage.
4. **No Physical Storage Details** – Doesn't explain database schemas.

Table I: Comparison Between Level 0 and Level 1 Table

Aspect	Level 0 DFD (Context)	Level 1 DFD (Detailed)
Scope	High-level overview of the system.	Internal breakdown of main processes.
Processes	Only 1 (Food Ordering System).	4 subprocesses (Order, Sold Items, Inventory, Report).
Entities	Customer, Kitchen, Manager.	Customer, Kitchen, Manager, Databases.
Detail Level	Abstract view.	More granular and detailed.
Purpose	Identify external interactions.	Show how system internally works.

6.2 Use Case Diagram

6.2.1 Introduction

A **Use Case Diagram** is a visual representation of the functional requirements of a system. It helps capture how different users (known as *actors*) interact with the system to achieve specific goals. In the context of the Online Food Ordering System (OrderEats), the Use Case Diagram provides a high-level overview of the services offered by the system to both customers and administrators.

Unlike technical diagrams such as DFD (Data Flow Diagram) or ERD (Entity Relationship Diagram), a Use Case Diagram focuses purely on what the system should do from the perspective of the user. It does not deal with how the processes are implemented, but instead highlights the user's needs, system boundaries, and interactions.

In modern software development, use case modeling is considered an essential tool during the requirement analysis phase. It ensures that all system functionalities are mapped correctly before moving into detailed system design and development.

6.2.2 Actors in the Online Food Ordering System:

In the Use Case Diagram of OrderEats, two primary actors have been identified:

1. **Customer (Primary Actor)**
   - Represents the end-user who interacts with the application to register, log in, browse menus, place orders, make payments, and track deliveries.
   - The customer is the main focus of the system, and most of the use cases are designed around providing them with a smooth and efficient food ordering experience.

2. **Admin (Secondary Actor)**
    - Represents the restaurant manager or administrator who ensures the backend operations of the system run smoothly.
    - The admin manages menus, updates food availability, monitors customer orders, and ensures that data such as prices, ingredients, and stock remain up to date.

These two actors cover both sides of the system: the **s**ervice consumers (customers) and the service providers (admins/restaurants).

6.2.3 Use Cases in the Online Food Ordering System:

The diagram illustrates multiple use cases (functionalities). Below is a detailed explanation of each:

1. Register

The registration process allows new customers to create an account within the application. They provide details such as name, email, phone number, and password. This ensures that only authenticated users can place and track orders.

- **Actors Involved:** Customer
- **Purpose:** Enable secure access to system features.
- **Precondition:** User must not already have an account.
- **Postcondition:** New account created successfully.

2. Login

The login functionality provides authenticated access for both customers and admins. Customers log in to browse menus and place orders, while admins log in to manage the backend system.

- **Actors Involved:** Customer, Admin
- **Purpose:** Authentication and authorization.
- **Precondition:** Valid credentials must exist.
- **Postcondition:** User gains access to the respective dashboard.

3. Navigate Menu

Customers can browse through the food items listed by the restaurant. The menu includes categories, food names, prices, descriptions, and images.

- **Actor Involved:** Customer
- **Purpose:** View and select available items.
- **Postcondition:** Menu displayed with filtering options (veg, non-veg, popular, price-based).

4. Select Item

Once the menu is displayed, customers can select specific food items they wish to order. Multiple items can be chosen in one session.

- **Actor Involved:** Customer
- **Purpose:** Choose desired food items.
- **Postcondition:** Selected items ready for adding to the cart.

5. Add to Cart

The selected items are placed into a temporary shopping cart. Customers can view, update, or remove items before finalizing the order.

- **Actor Involved:** Customer
- **Purpose:** Store selected items for checkout.
- **Postcondition:** Cart updated with items, prices, and quantities.

6. View Order

Customers can review their order before checkout. They can check quantities, pricing, and applicable taxes or discounts.

- **Actor Involved:** Customer
- **Purpose:** Verify order details.
- **Postcondition:** Order confirmed for further processing.

7. Proceed to Checkout

This functionality allows customers to move from the cart to the payment and delivery section. They can provide address details, phone number, and select a payment mode (e.g., UPI, card, or cash on delivery).

- **Actor Involved:** Customer
- **Purpose:** Confirm order and initiate payment.
- **Postcondition:** Order submitted to the system for processing.

8. Update Order

Customers may change order details before final confirmation. For example, they may increase/decrease quantities, add extra items, or cancel certain products.

- **Actor Involved:** Customer
- **Purpose:** Flexibility in order management.
- **Postcondition:** Order updated in the system database.

9. Receive Order (Admin)

The admin (restaurant staff) receives the placed order details for preparation and fulfillment. This ensures smooth coordination between the application and the restaurant.

- **Actor Involved:** Admin
- **Purpose:** Receive real-time order details.
- **Postcondition:** Order displayed on the admin dashboard.

10. Update Menu (Admin)

Admins can add new food items, update existing ones, modify prices, or mark unavailable dishes. This ensures that customers always see the latest menu.

- **Actor Involved:** Admin
- **Purpose:** Manage restaurant menu.
- **Postcondition:** Menu updated in the system and visible to customers.

11. Receive Confirmation

Once an order is placed and payment (if applicable) is processed, the system generates a confirmation message. The customer receives an update about the order status, expected delivery time, and tracking details.

- **Actor Involved:** Customer
- **Purpose:** Keep the user informed.
- **Postcondition:** Confirmation sent via app notification, SMS, or email.

12. Log Out

Both customers and admins can log out of the system once their tasks are complete. This ensures session security and prevents unauthorized access.

- **Actors Involved:** Customer, Admin
- **Purpose:** End user session securely.
- **Postcondition:** User redirected to login/landing page.

6.2.4 Importance of Use Case Diagram

The Use Case Diagram plays a vital role in system analysis and design. Its importance can be summarized as follows:

1. **Requirement Clarity** – Ensures that all required functionalities are clearly defined.
2. **User-Centric Design** – Focuses on the user's perspective rather than technical details.

3. **Improved Communication** – Acts as a bridge between developers, testers, and stakeholders.
4. **Early Error Detection** – Identifies missing functionalities early in the development phase.
5. **Supports Test Case Design** – Helps QA teams design scenarios for testing.

Figure V: Use Case Diagram

6.3 Entity Relationship (ER) Diagram

6.3.1 Introduction

An Entity Relationship (ER) Diagram is a conceptual data model that describes the structure of a database by defining the system's entities, their attributes, and relationships. For *OrderEats*, the ER diagram helps visualize how customers, admins, food items, orders, and payments are connected in the system.

The ER diagram ensures that the database design is clear, normalized, and optimized, providing a solid foundation for developing the application. It is particularly useful for identifying the logical structure before physical implementation in relational databases like MySQL or PostgreSQL.

6.3.2 Entities, Attributes, and Relationships

The given ER diagram contains five major entities:

1. **Customer**
2. **Order Details**
3. **Food**
4. **Payment**
5. **Admin**

Let's explain each in detail:

1. Customer Entity

The Customer entity represents the users of the system who interact with the app to browse menus, place orders, and make payments.

Attributes:

- **View ID**: A unique identifier for each customer record.
- **Name**: The full name of the customer.
- **Email**: Email ID used for communication and login.
- **Password**: Credential used for authentication and login security.

Importance:

- Customers are the primary end-users.
- Their data ensures personalized experiences, order history tracking, and secure access to the app.

Relationships:

- A Customer has Order Details (1:N relationship).
    - One customer can place multiple orders.
    - Each order is linked to only one customer.

2. Order Details Entity

The **Order Details** entity is central to the system as it captures all information about the transactions made by customers.

Attributes:

- **Order No.**: A unique identifier for each order.
- **Food Name**: The food item selected in the order.
- **Price**: The price of the ordered item(s).
- **Quantity**: The number of units ordered.
- **Phone No.**: Customer's phone number for contact and delivery.
- **Address**: Delivery location for the order.

Importance:

- It acts as a bridge between Customer, Food, and Payment.
- Maintains all necessary details to ensure correct delivery and billing.

Relationships:

- Order Details has Food (M:N relationship).
    - One order can contain multiple food items.
    - The same food item can appear in different orders.
- Order Details has Payment (N:1 relationship).
    - One order has exactly one payment transaction.
    - A payment entry corresponds to a single order.

3. Food Entity

The Food entity represents the menu items offered by the restaurant through the system.

Attributes:

- **Food ID**: A unique identifier for each food item.
- **Food Name**: Name of the dish.
- **Price**: Cost of the food item.
- **Description**: A short detail about the food, such as taste, category (veg/non-veg), or cuisine.
- **Ingredients**: List of key ingredients used to prepare the dish.

Importance:

- Defines what customers can browse, select, and order.
- Supports restaurant admins in managing menus dynamically.

Relationships:

- Food is added by Admin (N:1 relationship).

    - One admin can add multiple food items.
    - Each food item is linked to one admin for management.
- Food has Order Details (M:N relationship).
    - A food item can appear in multiple orders.
    - Each order can include multiple food items.

4. Payment Entity

The **Payment** entity handles all monetary transactions within the system.

Attributes:

- **Payment ID**: Unique identifier for each transaction.
- **Payment Name**: Title or reference for the transaction.
- **Payment Mode**: Mode of payment (UPI, card, wallet, COD).
- **Price**: The total bill amount paid.

Importance:

- Ensures that transactions are tracked and stored securely.
- Acts as proof of purchase for both the restaurant and the customer.

Relationships:

- Payment has Order Details (1:N relationship).
    - One payment corresponds to one order.
    - An order cannot exist without a payment record.

5. Admin Entity

The Admin entity represents the restaurant's management personnel responsible for adding and maintaining food items in the system.

Attributes:

- **Admin ID**: Unique identifier for each admin.
- **Admin Name**: Full name of the admin.
- **Admin Password**: Credential for authentication.
- **Restaurant Name**: The restaurant managed by the admin.

Importance:

- Provides control over the menu and prices.
- Ensures data consistency by allowing only authorized personnel to update food details.

Relationships:

- Admin adds Food (1:N relationship).
    - One admin can add multiple food items.
    - Each food item is linked to exactly one admin.

6.3.3 Key Relationships and Their Cardinality

1. **Customer → Order Details →** (1:N)
    - A customer can place multiple orders.
2. **Order Details → Food →** (M:N)
    - An order can include multiple food items, and a food item can belong to multiple orders.
3. **Order Details → Payment →** (N:1)
    - Each order has exactly one payment, but one payment belongs to only one order.
4. **Admin → Food →** (1:N)
    - An admin can add multiple food items to the system.

6.3.4 Normalization and Database Design Benefits

The ER diagram ensures the database follows normalization rules, preventing redundancy and maintaining data integrity:

- **1NF (First Normal Form)**: Every attribute is atomic (e.g., Food Name, Price).
- **2NF (Second Normal Form)**: Non-key attributes depend fully on primary keys (e.g., Food Description depends on Food ID).
- **3NF (Third Normal Form)**: No transitive dependency exists (e.g., Payment attributes depend only on Payment ID).

6.3.5 Advantages of the ER Model

1. **Clarity of Database Design** – Simplifies complex relationships into a visual model.
2. **Data Integrity** – Ensures valid relationships and prevents inconsistencies.
3. **Scalability** – Easy to add new entities like “Delivery Partner” or “Offers”.
4. **Efficient Queries** – Helps optimize queries for orders, payments, and menu management.
5. **Separation of Roles** – Customers, Admins, and Payments are clearly distinguished.

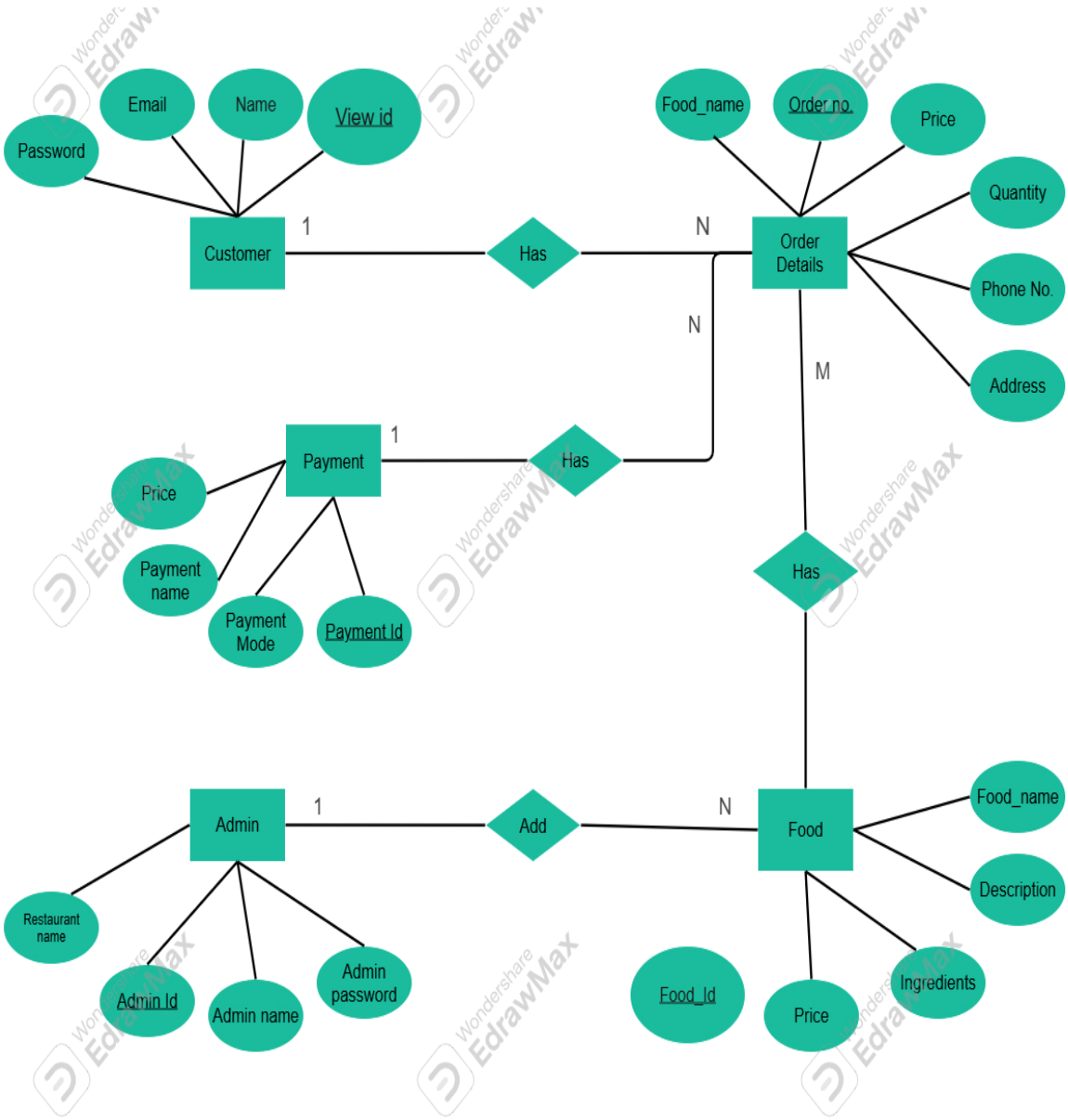


Figure VI: ER Diagram

6.4 Input and Output Screen Design

The OrderEats app prioritizes a clean, user-friendly design to make ordering food quick and intuitive.

Input Screens:

1. **Registration/Login** – Fields for name, email, password, and location.
2. **Location Selection** – GPS-based or manual entry.
3. **Search & Browse** – Filters for cuisine, price, and category.
4. **Cart** – Editable list of items with quantity controls.

5. **Payment Page** – Options for UPI, Phone Pay, Paytm, GPay, Etc

Output Screens:

1. **Order Confirmation** – Displays order ID, summary, and delivery estimate.
2. **Receipt** – Downloadable invoice with details.
3. **Order History** – List of past orders with statuses.
4. **Profile Dashboard** – User info and preferences.

App Screen Flow – OrderEats

The App Screen Flow of OrderEats is designed to provide users with a seamless, step-by-step journey from launching the app to successfully placing and tracking food orders. The flow prioritizes simplicity, clarity, and intuitive navigation, ensuring that even first-time users can operate the app with minimal learning effort.

The journey begins with the Splash Screen, which displays the logo while initializing resources in the background. Users are then directed to the Start Screen, where they can either log in or register. Returning users proceed directly to the Login Screen, while new users move to the Sign-Up Screen to create an account by entering details such as name, email, and password.

After successful authentication, the Choose Location Screen is presented. Here, the user may select their delivery address using GPS auto-detection or manual entry. Once the location is set, the app transitions to the Main Screen, which is built around a bottom navigation bar hosting key fragments: Home, Search, Cart, History, and Profile.

- **Home Fragment**: Displays popular dishes, promotions, and featured restaurants.
- **Search Fragment**: Allows filtering and keyword-based search for menu items.
- **Cart Fragment**: Provides a detailed view of items added, with options to adjust quantity or remove items before checkout.
- **History Fragment**: Lists past orders with status updates, receipts, and reorder options.
- **Profile Fragment**: Lets users update personal information, delivery addresses, and preferences.

During checkout, the user is directed to a Payment Screen, where they choose between UPI, card, or cash on delivery. Upon confirmation, an Order Receipt Screen displays the order summary and estimated delivery time.

This carefully structured screen flow ensures a logical progression that reduces errors, enhances usability, and builds user confidence in the system.

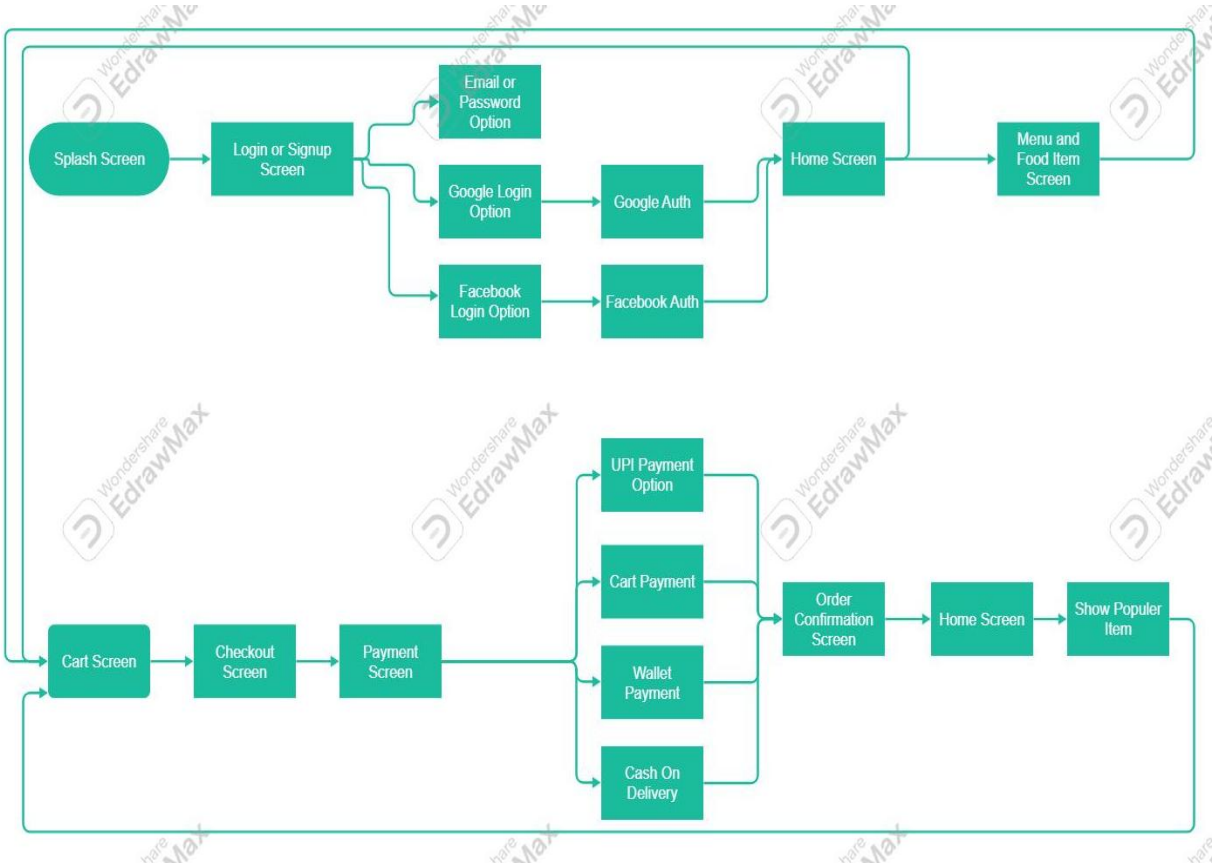


Figure VII: App Screen Flow – OrderEats

6.5 Processes Involved

- Registration/Login: Validate inputs; create or verify user credentials.
- Location Selection: Acquire GPS with permission; else accept manual entry.
- Browsing & Search: Filter items by category or search query; show ratings and prices.
- Cart & Checkout: Add items, update quantities, compute totals, and submit order.
- History & Profile: Persist past orders and allow profile edits.

6.6 Methodology of Testing

Testing is a crucial phase in the software development life cycle, ensuring that the application functions as expected, meets user requirements, and is free from defects. The methodology of testing defines the structured approach used to validate the software and verify that it meets its intended purpose. In this project, multiple testing strategies are employed to cover different aspects of the system, including unit testing, integration testing, UI testing, and acceptance testing. Each of these strategies serves a specific purpose and collectively ensures the quality and reliability of the application.

6.6.1 Importance of Testing

Software testing is not just about finding errors; it is about improving the quality of the software, ensuring maintainability, and reducing future costs associated with defects. Testing provides confidence to developers, stakeholders, and users that the application behaves correctly under various conditions. A well-defined testing methodology also helps in identifying potential issues early in the development process, which is more efficient and cost-effective than addressing problems after deployment. Furthermore, rigorous testing ensures that changes or updates to the software do not introduce new errors, which is particularly important in agile development environments where continuous integration and frequent releases are common.

6.6.2 Unit Testing

Unit testing is the process of verifying individual components or modules of the software in isolation to ensure that each part works correctly. In this project, unit testing is applied to adapters and utility classes, such as price calculators, data parsers, and helper functions. These classes are tested using local unit tests to confirm that their logic produces the expected results for various input values. For instance, the price calculator utility is tested with different combinations of items, discounts, and taxes to ensure accurate calculations.

Unit tests are automated, allowing developers to quickly run tests whenever code changes are made. This automation ensures that regressions are detected immediately, reducing the risk of introducing bugs into the system. Unit testing frameworks like JUnit for Java or Kotlin provide features such as assertions, test runners, and mocking, which help simulate different scenarios and validate the behavior of individual units. By thoroughly testing each component at this level, the development team can ensure that the foundation of the application is stable and reliable.

6.6.3 Integration Testing

While unit testing focuses on individual modules, integration testing examines how different modules interact with each other. In this project, integration testing is used to validate navigation flows across Activities and Fragments, ensuring that data is passed correctly, and user actions trigger the appropriate responses. Tools such as Espresso are employed to automate these tests in Android applications, simulating user interactions like button clicks, text input, and screen transitions.

Integration tests are critical because even if individual modules work correctly, errors may arise when they are combined. For example, a user logging in may navigate from the login Activity to the home Fragment. Integration tests verify that the user data is transferred accurately, the UI updates as expected, and no crashes occur during the transition. This testing also covers error handling scenarios, such as network failures, empty input fields, or invalid credentials, to ensure the system behaves gracefully under unexpected conditions.

6.6.4 UI Testing

User Interface (UI) testing focuses on evaluating the look, feel, and usability of the application. It ensures that the application meets design specifications, provides a smooth user experience, and handles edge cases correctly. In this project, UI testing is performed manually through exploratory testing. Testers interact with the application as real users would, exploring different screens, performing actions, and verifying that the interface responds appropriately.

UI testing also includes testing rotation changes, such as switching from portrait to landscape mode, to confirm that the layout remains consistent and functional. Edge cases, like entering extremely long text, invalid characters, or unexpected input sequences, are checked to prevent crashes or misbehaviour. While automated UI testing can be implemented, manual exploratory testing is valuable because it allows testers to discover unexpected issues that automated scripts might miss, such as visual misalignments or confusing navigation flows.

6.6.5 Acceptance Testing

Acceptance testing validates the software against user requirements and business goals. It ensures that the system fulfils the intended purpose and is ready for deployment. In this project, acceptance testing involves running end-to-end order scenarios with sample data. Testers simulate the complete user journey, from signing up and logging in to placing an order, making payments, and receiving notifications. Feedback is

collected at each stage to identify areas for improvement, verify system reliability, and ensure the application meets user expectations.

Acceptance tests are often performed in collaboration with stakeholders or end users to confirm that the software delivers value and aligns with business objectives. This testing phase is particularly important for validating workflow correctness, data accuracy, and overall system usability. It also helps detect gaps in functionality, identify missing features, and confirm that the system is ready for production deployment.

6.6.6 Testing Tools and Frameworks

Various tools and frameworks support the testing methodology applied in this project. For unit testing, frameworks such as JUnit, Mockito, or Robolectric are used to write and run automated tests for individual components. For integration and UI testing, Espresso provides a robust environment to simulate user interactions and verify activity and fragment transitions. These tools allow for reproducible, automated, and efficient testing, ensuring that the software maintains high quality even as it evolves.

Additionally, manual testing techniques complement automated testing by exploring areas that automated scripts cannot fully cover. Test cases are documented, executed, and reviewed systematically to ensure comprehensive coverage. Bug tracking tools like Jira or GitHub Issues help log defects, track progress, and prioritize fixes, making the testing process organized and transparent.

6.6.7 Test Case Design

Effective test case design is essential to ensure that testing is thorough and covers all critical scenarios. Test cases are written to cover positive scenarios (expected usage), negative scenarios (invalid input or unexpected actions), and boundary cases (edge conditions). For example, test cases for the order functionality may include placing a single item order, multiple item orders, orders with discounts, invalid payment details, and network failures.

Each test case specifies the input conditions, expected output, and actual results, providing a clear basis for evaluating the system’s behavior. This structured approach helps testers systematically verify all aspects of the application and ensures that defects are detected early and efficiently.

6.6.8 Benefits of the Testing Methodology

The structured testing methodology applied in this project provides multiple benefits:

1. Improved Software Quality: By validating individual components, their interactions, and overall system behavior, the software is more reliable and less prone to defects.
2. Early Defect Detection: Unit and integration testing help identify issues early in the development process, reducing costs and effort required for fixes.
3. User Satisfaction: Acceptance and UI testing ensure that the application meets user expectations, providing a smooth and intuitive experience.
4. Maintainability: Well-tested code is easier to maintain and extend because developers can confidently make changes without introducing new defects.
5. Risk Mitigation: Comprehensive testing reduces the risk of critical failures in production, ensuring business continuity and customer trust.

6.6.9 Challenges in Testing

Despite its importance, software testing faces certain challenges. Automated tests require proper setup and maintenance, and they may not capture all user experience issues. Manual testing can be time-consuming and may vary depending on tester expertise. Moreover, testing complex systems with multiple dependencies requires careful planning to ensure adequate coverage without excessive effort. Balancing these challenges while maintaining high-quality testing is a critical aspect of the methodology.

Table II: Sample Test Case Table

Test ID	Feature Tested	Input	Expected Output	Status
T01	Login	Valid email & password	Redirect to Home	Pass
T02	Login	Wrong password	Show error message	Pass
T03	Add to Cart	Item = Burger, Qty = 2	Cart shows total for 2 burgers	Pass
T04	Checkout	Valid address + payment	Order confirmed	Pass
T05	Order History	Retrieve past orders	Display list	Pass

6.7 Coding

6.7.1 Screenshots of User Interface Code

6.7.1.1 Splash_Screen.kt

splash_screen.kt

```kotlin
package com.rajender.ordereats

import ...

// Rename your class to follow Kotlin conventions (PascalCase)
// Also, the @Suppress("DEPRECATION") might not be needed if Handler(Looper) is used.
@SuppressLint("CustomSplashScreen") // It's good practice to acknowledge this
class SplashScreenActivity : AppCompatActivity() { // 2. Rename class

    private lateinit var binding: ActivitySplashScreenBinding // 3. Declare binding variable

    // Animation Delays (can be adjusted)
    private val LOGO_ANIM_DELAY = 100L
    private val TITLE_ANIM_DELAY = 400L // Stagger after logo
    private val SUBTITLE_ANIM_DELAY = 700L // Stagger after title
    private val DEV_NAME_ANIM_DELAY = 1000L // Stagger after subtitle

    // Total time splash screen is visible. Should be long enough for animations + a brief view.
    // Make sure this is longer than your longest animation sequence.
    private val SPLASH_DISPLAY_LENGTH: Long = 2800 // Adjusted from 1000ms

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge() // Keep if you use edge-to-edge

        // 4. Initialize ViewBinding
        binding = ActivitySplashScreenBinding.inflate(layoutInflater)
        setContentView(binding.root) // Use binding.root

```

6.7.1.2. LoginActivity.kt

LoginActivity.kt

```kotlin
package com.rajender.ordereats

// import com.google.firebase.database.FirebaseDatabase // Only if specifically needed here
import ...

class LoginActivity : AppCompatActivity() {

    private lateinit var binding: ActivityLoginBinding
    private lateinit var auth: FirebaseAuth
    // private lateinit var database: FirebaseDatabase // Only initialize if you use it in LoginActivity
    private lateinit var googleSignInClient: GoogleSignInClient
    private lateinit var googleSignInLauncher: ActivityResultLauncher<Intent>

    // Define delays for staggering
    companion object { // Using companion object for constants
        private const val TAG = "LoginActivity" // For Logging
        private const val DELAY_LOGO = 100L
        private const val DELAY_APP_NAME = DELAY_LOGO + 200L
        private const val DELAY_TAGLINE = DELAY_APP_NAME + 200L
        private const val DELAY_LOGIN_PROMPT = DELAY_TAGLINE + 200L
        private const val DELAY_EMAIL_FIELD = DELAY_LOGIN_PROMPT + 250L
        private const val DELAY_PASSWORD_FIELD = DELAY_EMAIL_FIELD // Can start simultaneously or slightly
        private const val DELAY_OR_TEXT = DELAY_PASSWORD_FIELD + 300L
        private const val DELAY_CONTINUE_TEXT = DELAY_OR_TEXT // Converging, start at same time as "Or"
        private const val DELAY_GOOGLE_BUTTON = DELAY_OR_TEXT + 250L
        private const val DELAY_FACEBOOK_BUTTON = DELAY_GOOGLE_BUTTON // Can start simultaneously
        private const val DELAY_LOGIN_BUTTON = DELAY_GOOGLE_BUTTON + 300L
        private const val DELAY_SIGNUP_PROMPT = DELAY_LOGIN_BUTTON + 200L
        private const val DELAY_DEV_NAME = DELAY_SIGNUP_PROMPT + 300L
```

6.7.1.3. SignupActivity.kt

SignActivity.kt

```kotlin
package com.rajender.ordereats

import ...

class SignActivity : AppCompatActivity() {

    private var inputUsername: String = ""
    private var inputEmail: String = ""
    private var inputPassword: String = ""
    private lateinit var auth: FirebaseAuth
    private lateinit var database: DatabaseReference
    private lateinit var googleSignInClient: GoogleSignInClient
    private val binding: ActivitySignBinding by lazy {
        ActivitySignBinding.inflate(layoutInflater)
    }
    // Animation Delays
    private val DELAY_LOGO = 100L
    private val DELAY_APP_NAME = DELAY_LOGO + 150L
    private val DELAY_TAGLINE = DELAY_APP_NAME + 150L
    private val DELAY_SIGNUP_PROMPT = DELAY_TAGLINE + 150L
    private val DELAY_NAME_FIELD = DELAY_SIGNUP_PROMPT + 200L
    private val DELAY_EMAIL_FIELD = DELAY_NAME_FIELD + 100L
    private val DELAY_PASSWORD_FIELD = DELAY_EMAIL_FIELD + 100L
    private val DELAY_OR_TEXT = DELAY_PASSWORD_FIELD + 250L
    private val DELAY_SIGNUP_WITH_TEXT = DELAY_OR_TEXT
    private val DELAY_GOOGLE_BUTTON = DELAY_OR_TEXT + 200L
    private val DELAY_FACEBOOK_BUTTON = DELAY_GOOGLE_BUTTON
    private val DELAY_CREATE_ACCOUNT_BUTTON = DELAY_GOOGLE_BUTTON + 250L
    private val DELAY_ALREADY_HAVE_ACCOUNT_PROMPT = DELAY_CREATE_ACCOUNT_BUTTON + 200L
```

6.7.1.4. ChooseLocationActivity.kt

ChooseLocationActivity.kt

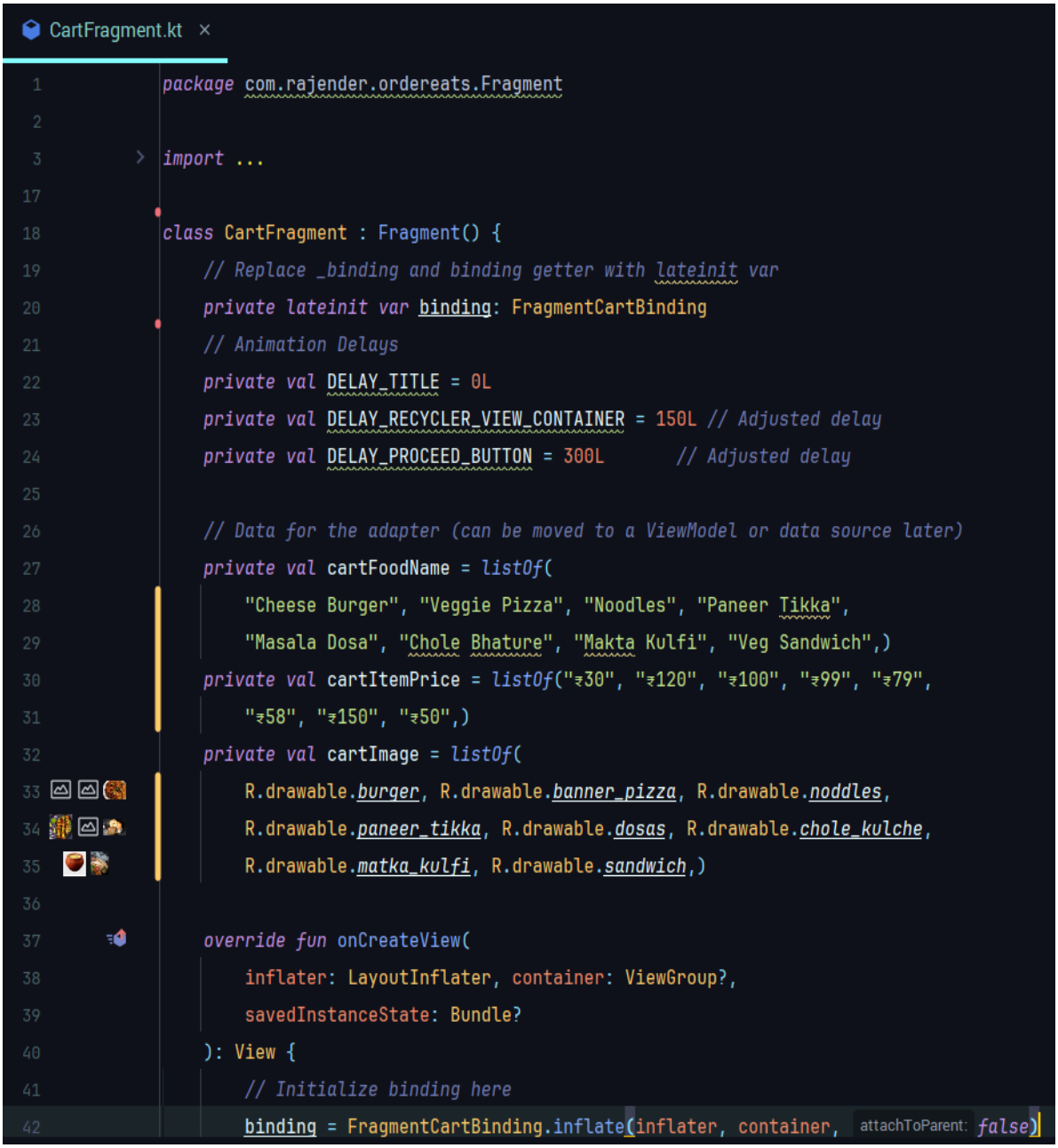
```kotlin
package com.rajender.ordereats

// Removed: import android.R
import ...

class ChooseLocationActivity : AppCompatActivity() {
    private val binding : ActivityChooseLocationBinding by lazy {
        ActivityChooseLocationBinding.inflate(layoutInflater)
    }
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContentView(binding.root)
        val locationList = arrayOf("Jaipur", "Delhi", "Uttar-Pradesh", "Odisha", "Punjab", "Haryana")
        // Corrected line:
        val adapter = ArrayAdapter( context: this, android.R.layout.simple_list_item_1, locationList)

        val autoCompleteTextView: AutoCompleteTextView = binding.listOfLocation
        autoCompleteTextView.setAdapter(adapter)

        // It's good practice to also enable the dropdown to show on click
        // if it's meant to behave like a spinner
        autoCompleteTextView.isFocusable = false
        autoCompleteTextView.isClickable = true
        autoCompleteTextView.setOnClickListener {
            autoCompleteTextView.showDropDown()
        }
    }
}
```

6.7.1.5 MainActivity.kt

MainActivity.kt

```kotlin
package com.rajender.ordereats

import ...

class MainActivity : AppCompatActivity() {

    // Using lateinit for View Binding as it's initialized in onCreate
    private lateinit var binding: ActivityMainBinding
    private lateinit var navController: NavController

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        // It's generally good practice to inflate binding before setContentView
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)
        // 1. Get the NavHostFragment using its ID
        val navHostFragment =
            supportFragmentManager.findFragmentById(R.id.fragmentContainerView) as NavHostFragment

        // 2. Get the NavController from the NavHostFragment
        navController = navHostFragment.navController

        // 3. Setup BottomNavigationView with NavController
        // Using view binding to access bottomNavigationView for safety and conciseness
        binding.bottomNavigationView.setupWithNavController(navController)

        binding.notificationButton.setOnClickListener {
            // Consider using a more descriptive tag or a companion object constant for the tag
            val bottomSheetDialog = Notification_Botton_Fragment()
```

6.7.1.6 CartFragment.kt



6.7.1.7 SearchFragment.kt

SearchFragment.kt

```kotlin
package com.rajender.ordereats.Fragment

import ...

class SearchFragment : Fragment() {
    // Using _binding pattern for safer view access
    private var _binding: FragmentSearchBinding? = null
    private val binding get() = _binding!!

    private lateinit var adapter: MenuAdapter // Keep this if your MenuAdapter is designed this way

    // Original data - consider making these immutable if they don't change after init
    private val originalMenuFoodName = listOf("Veggie Pizza",
        "Noodles",
        "Paneer Tikka",
        "Masala Dosa",
        "Chole Bhature",
        "Makta Kulfi",
        "Veg Sandwich",
        "Momo Platter",
        "Ice Cream",
        "Kachori",
        "Desi Jalebi",
        "Mojito",
        "Manchurian",
        "Desi Aalu Paratha")
    private val originalMenuItemPrice = listOf("₹30", "₹120", "₹100", "₹99", "₹79", "₹58", "₹150", "₹50",
    private val originalMenuImages = listOf(
        R.drawable.burger,
```

6.7.1.8 HistoryFragment.kt

HistoryFragment.kt

```kotlin
package com.rajender.ordereats.Fragment

import ...

class HistoryFragment : Fragment() {
    // Using _binding pattern for safer view access
    private var _binding: FragmentHistoryBinding? = null
    private val binding get() = _binding!!

    private lateinit var buyAgainAdapter: BuyAgainAdapter // Keep this
    // Data for the RecyclerView (can be kept as is for this example)
    private val buyAgainFoodName = arrayListOf(
        "Cheese Burger", "Veggie Pizza", "Noodles", "Paneer Tikka",
        "Masala Dosa", "Chole Bhature", "Makta Kulfi", "Veg Sandwich",
        "Momo Platter", "Ice Cream", "Kachori", "Desi Jalebi", "Mojito",
        "Manchurian", "Desi Aalu Paratha")
    private val buyAgainFoodPrice = arrayListOf("₹30", "₹120", "₹100", "₹99",
        "₹79", "₹58", "₹150", "₹50", "₹110","₹99","₹30","₹50","₹50","₹70","₹40")
    private val buyAgainFoodImage = arrayListOf(
        R.drawable.burger,
        R.drawable.banner_pizza,
        R.drawable.noddles,
        R.drawable.paneer_tikka, // Replace with your actual drawables
        R.drawable.dosas,
        R.drawable.chole_kulche,
        R.drawable.matka_kulfi,
        R.drawable.sandwich,
        R.drawable.momos_2,
        R.drawable.ice_cream,
```

6.7.1.9 ProfileFragment.kt

ProfileFragment.kt

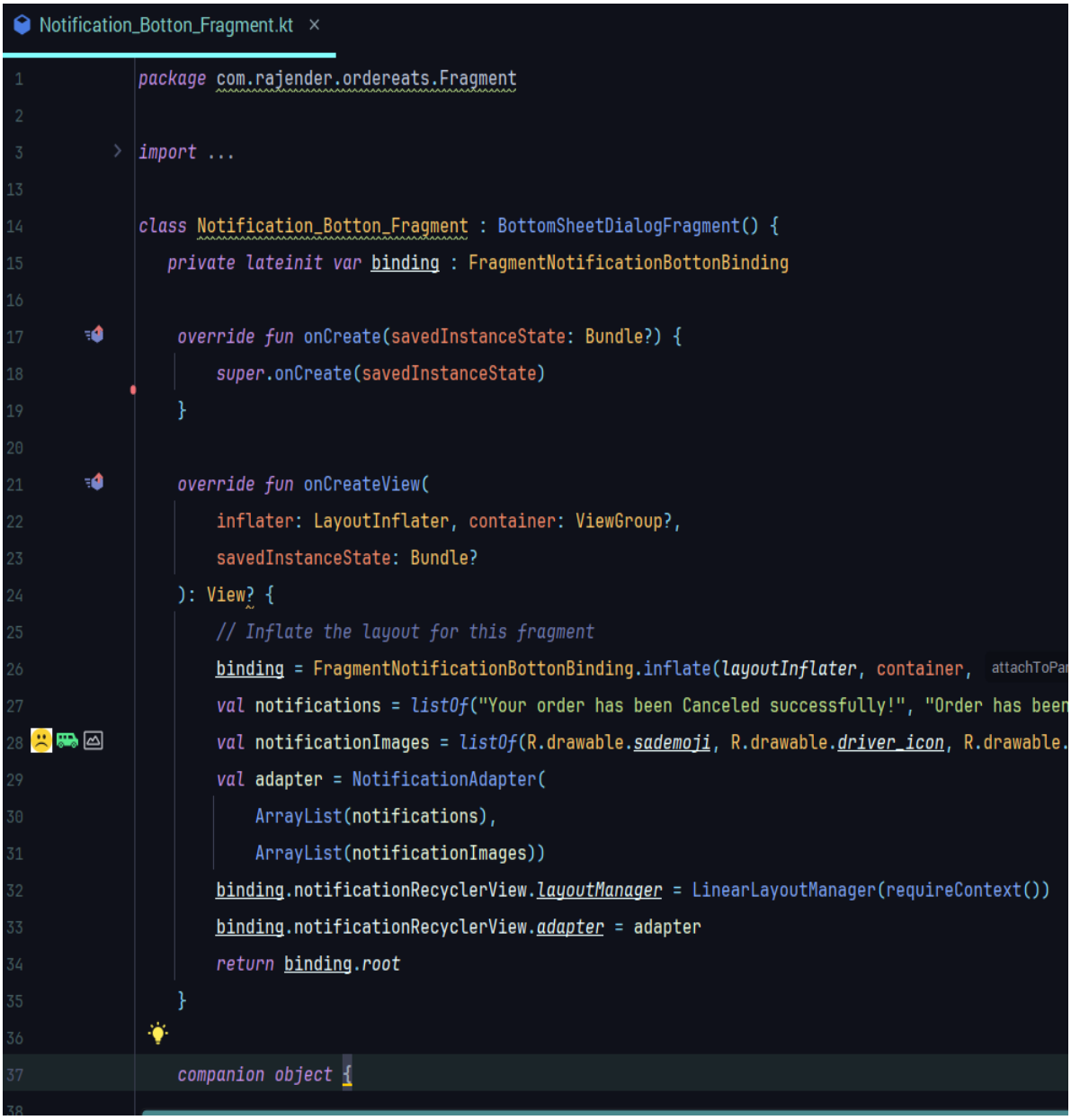
```kotlin
package com.rajender.ordereats.Fragment

import ...

class ProfileFragment : Fragment() {
    // Use ViewBinding
    private var _binding: FragmentProfileBinding? = null
    // This property is only valid between onCreateView and onDestroyView.
    private val binding get() = _binding!!

    // Animation Delays
    private val DELAY_FIELD_INITIAL = 50L // Start a bit later for smoother perceived
    private val DELAY_FIELD_INCREMENT = 120L // Stagger delay between fields
    // Calculate delay for save button to appear after the last field
    private val DELAY_SAVE_BUTTON = DELAY_FIELD_INITIAL + (DELAY_FIELD_INCREMENT * 3)

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
    }

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        _binding = FragmentProfileBinding.inflate(inflater, container,  attachToParent: fa
        return binding.root
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
```

6.7.1.10 MenuBottomSheetFragment.kt

menuBottomSheetFragment.kt

```kotlin
package com.rajender.ordereats.Fragment // Ensure this is your correct package

import ...

class MenuBottomSheetFragment : BottomSheetDialogFragment() {
    private lateinit var binding: FragmentMenuBottomSheetBinding

    private val menuFoodNames = listOf(
        "Cheese Burger", "Veggie Pizza", "Noodles", "Paneer Tikka", "Masala Dosa",
        "Chole Bhature", "Makta Kulfi", "Veg Sandwich", "Momo Platter", "Ice Cream",
        "Kachori", "Desi Jalebi", "Mojito", "Manchurian", "Desi Aalu Paratha"
    )
    private val menuItemPrices = listOf(
        "₹30", "₹120", "₹100", "₹99", "₹79", "₹58", "₹150", "₹50", "₹110",
        "₹99", "₹30", "₹50", "₹50", "₹70", "₹40"
    )
    private val menuImages = listOf(
        R.drawable.burger, R.drawable.banner_pizza, R.drawable.noddles,
        R.drawable.paneer_tikka, R.drawable.dosas, R.drawable.chole_kulche,
        R.drawable.matka_kulfi, R.drawable.sandwich, R.drawable.momos_2,
        R.drawable.ice_cream, R.drawable.kachori, R.drawable.jalebi,
        R.drawable.mojito, R.drawable.manchurian, R.drawable.paratha
    )

    // Animation Delays (in milliseconds)
    private val DELAY_BACK_BUTTON = 50L
    private val DELAY_TITLE = 150L
    private val DELAY_RECYCLER_VIEW_CONTAINER = 250L
```

6.7.1.11 NotificationButtonFragment.kt



6.7.1.12 DetailsActivity.kt

DetailsActivity.kt

```kotlin
package com.rajender.ordereats

import ...

class DetailsActivity : AppCompatActivity() {
    private lateinit var binding: ActivityDetailsBinding
    // Animation Delays (in milliseconds)
    private val DELAY_BACK_BUTTON = 100L
    private val DELAY_FOOD_NAME = 200L
    private val DELAY_FOOD_IMAGE_CARD = 300L
    private val DELAY_DESC_TITLE = 400L
    private val DELAY_DESC_TEXT = 500L
    private val DELAY_INGREDIENTS_TITLE = 600L
    private val DELAY_INGREDIENTS_TEXT = 700L
    private val DELAY_ADD_TO_CART_BUTTON = 800L

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityDetailsBinding.inflate(layoutInflater)
        setContentView(binding.root)

        val slideInFromLeft = AnimationUtils.loadAnimation( context: this, R.anim.slide_in_from_left_simple)
        val fadeInText = AnimationUtils.loadAnimation( context: this, R.anim.fade_in_text)
        val scaleFadeInCard = AnimationUtils.loadAnimation( context: this, R.anim.scale_fade_in_card)
        val slideUpFromBottom = AnimationUtils.loadAnimation( context: this, R.anim.slide_up_from_bottom)
        val clickScale = AnimationUtils.loadAnimation( context: this, R.anim.click_scale)

        val viewsToAnimate = listOf(
            binding.imageButton,
```

6.7.1.13 PayOutActivity.kt

PayOutActivity.kt

```kotlin
package com.rajender.ordereats

import ...

class PayOutActivity : AppCompatActivity() {
    private lateinit var binding: ActivityPayOutBinding
    private var selectedPaymentMethod: String = "Cash On Delivery"

    private val DELAY_BACK_BUTTON = 100L
    private val DELAY_TITLE = 200L
    private val DELAY_NAME_FIELD = 300L
    private val DELAY_ADDRESS_FIELD = 370L
    private val DELAY_PHONE_FIELD = 440L
    private val DELAY_PAYMENT_TITLE = 510L // Ensure this corresponds to a view if used
    private val DELAY_RADIO_GROUP = 580L
    private val DELAY_TOTAL_FIELD = 650L
    private val DELAY_PLACE_ORDER_BUTTON = 750L

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityPayOutBinding.inflate(layoutInflater)
        setContentView(binding.root)

        setupAnimations()

        binding.buttonBack.setOnClickListener {
            finish()
        }
```

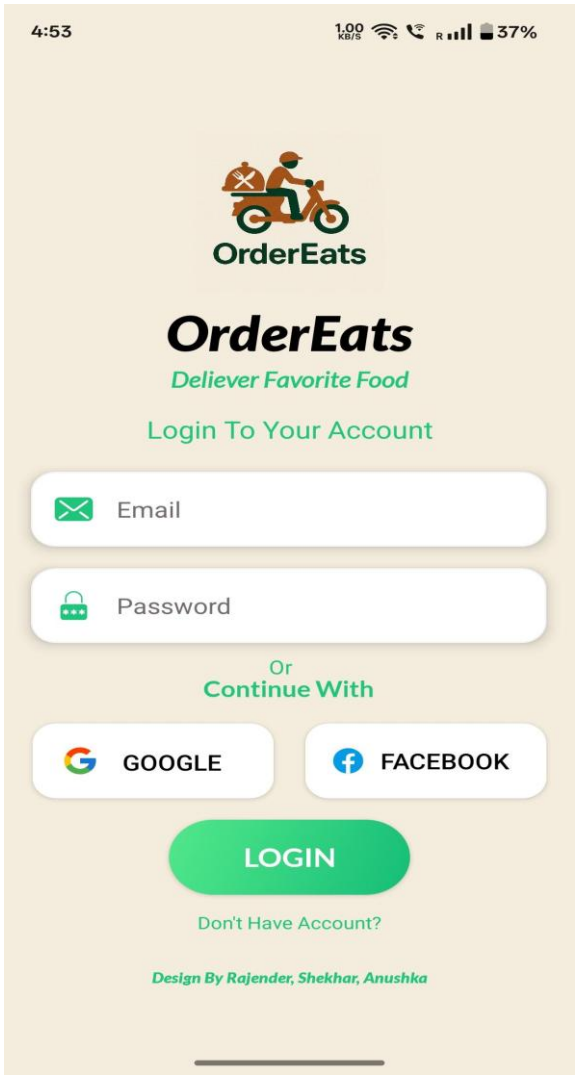
6.7.1.14 CongratsFragment.kt

CongratsBottomSheet.kt

```
1   package com.
                C:\Users\vkver\OneDrive\Desktop\Apps\OrderEats_App\OrderEats\app\src\main\java\com\rajender\ordereats\Fragment
2
3   import ...
15
16  class CongratsBottomSheet : BottomSheetDialogFragment() {
17      private lateinit var binding: FragmentCongratsBottomSheetBinding
18      // Animation Delays
19      private val DELAY_TEXT = 100L
20      private val DELAY_IMAGE = 250L
21      private val DELAY_BUTTON = 450L
22
23      override fun onCreate(savedInstanceState: Bundle?) {
24          super.onCreate(savedInstanceState)
25      }
26
27      override fun onCreateView(
28          inflater: LayoutInflater, container: ViewGroup?,
29          savedInstanceState: Bundle?
30      ): View { // Return type should be non-nullable View
31          binding = FragmentCongratsBottomSheetBinding.inflate(inflater, container,  attachToParent: false)
32          return binding.root
33      }
34
35      override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
36          super.onViewCreated(view, savedInstanceState)
37          // --- 1. Load Animations ---
38          val fadeInScaleTextAnim = AnimationUtils.loadAnimation(requireContext(), R.anim.fade_in_scale_text)
39          val scaleBounceImageAnim = AnimationUtils.loadAnimation(requireContext(), R.anim.scale_bounce_fade_in_
40          val slideUpButtonAnim = AnimationUtils.loadAnimation(requireContext(), R.anim.slide_up_fade_in_button)
```

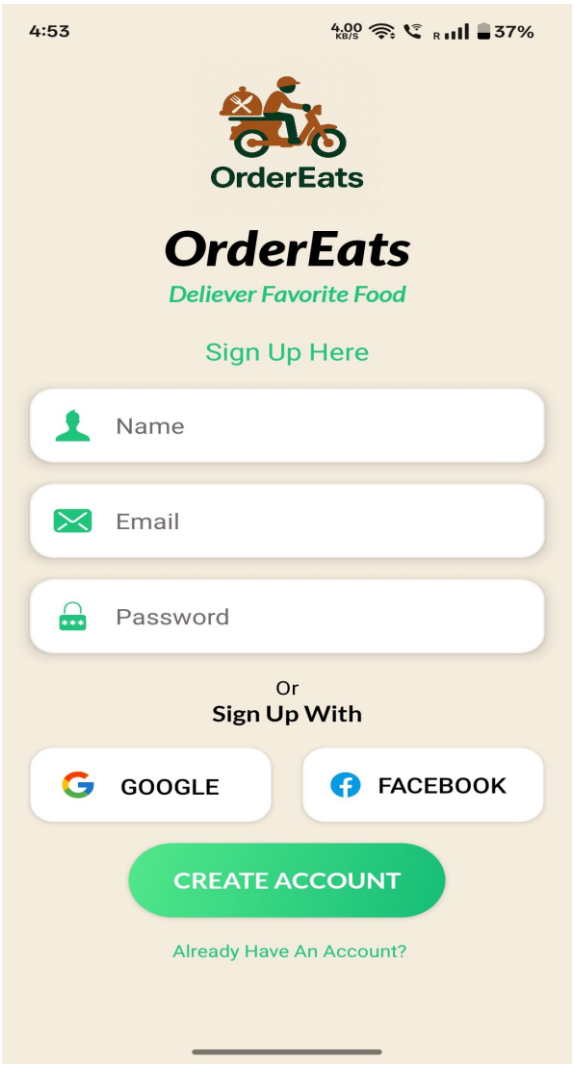
6.7.2 User Interface Screenshots of the project

6.7.2.1 Login Page



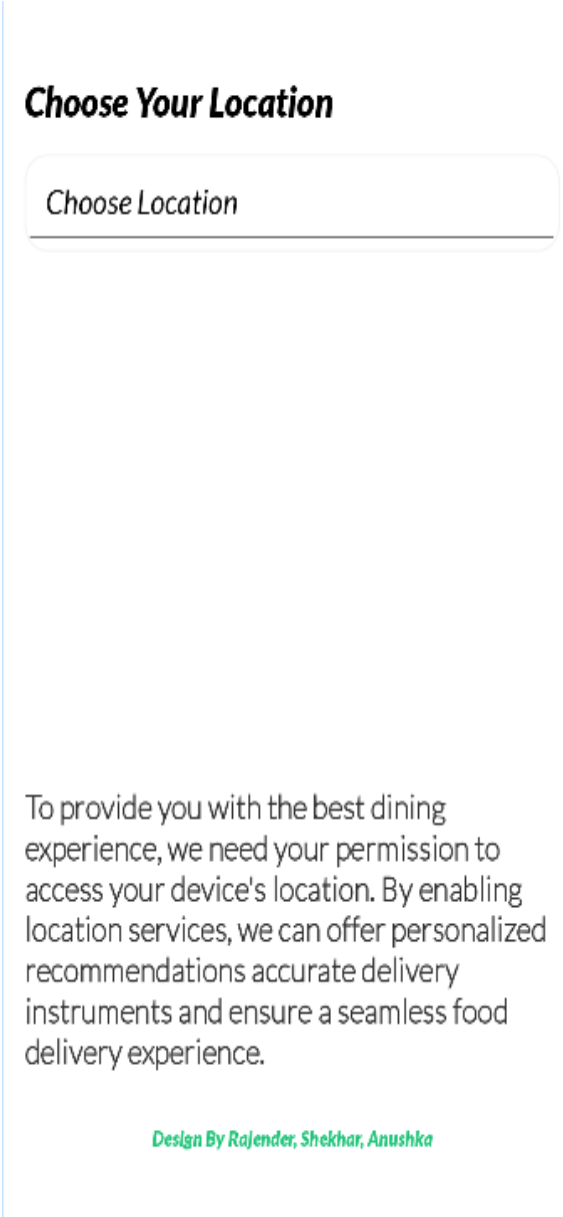
The **Login Page** allows users to sign in securely. It ensures only authorized access.

6.7.2.2 Sign-Up page



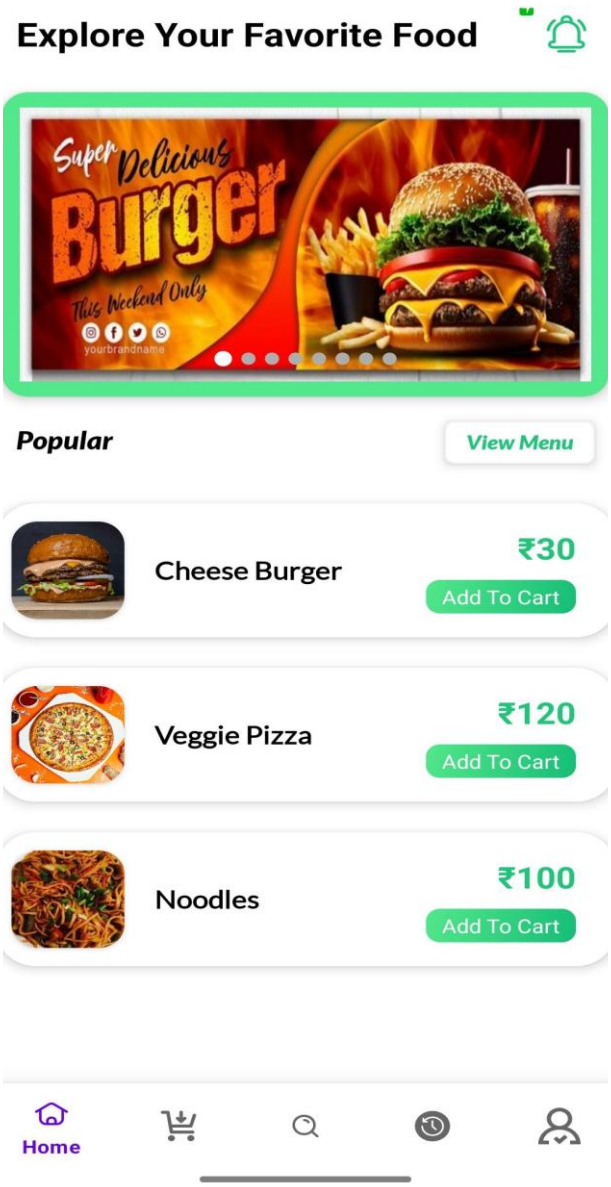
The **Sign Up Page** enables new users to create an account. It collects basic details for registration.

6.7.2.3 Choose Location Page



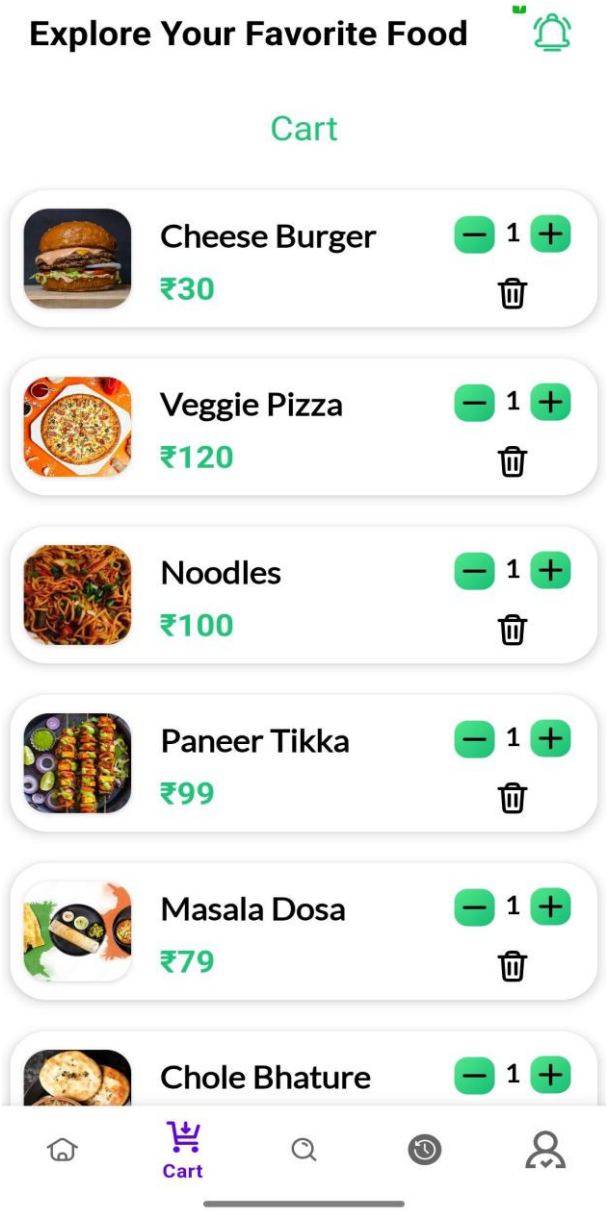
The **Choose Location Page** allows users to set their delivery address. It supports GPS detection or manual entry.

6.7.2.4 Main Page



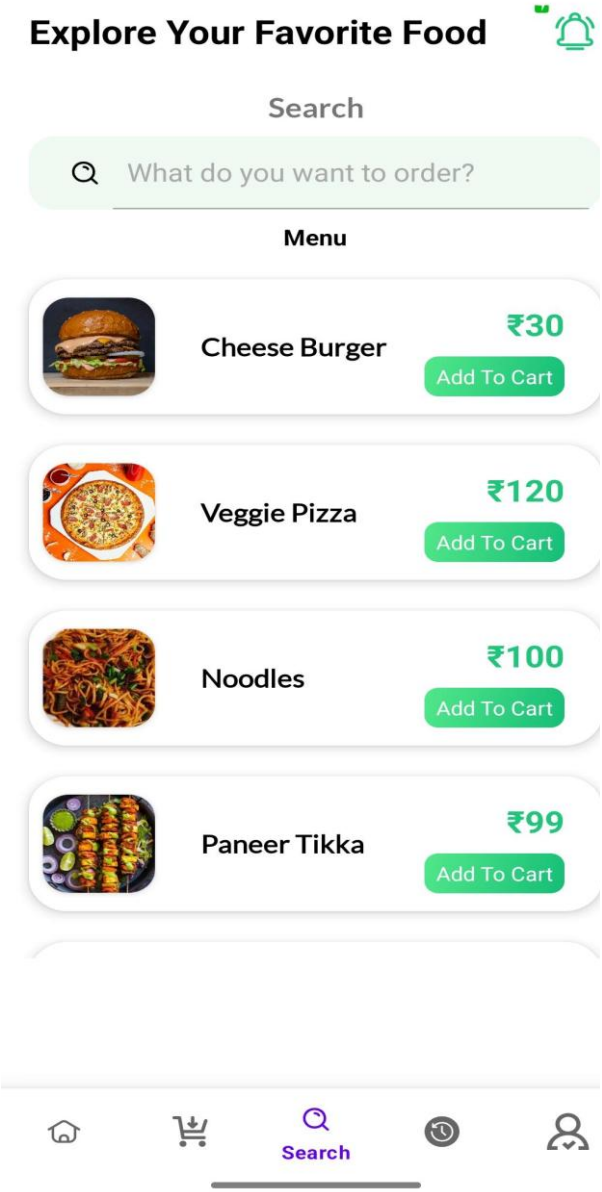
The **Main Page** serves as the dashboard, showing navigation options. Users can access Home, Cart, Profile, Search, and History from here.

6.7.2.5 Cart Page



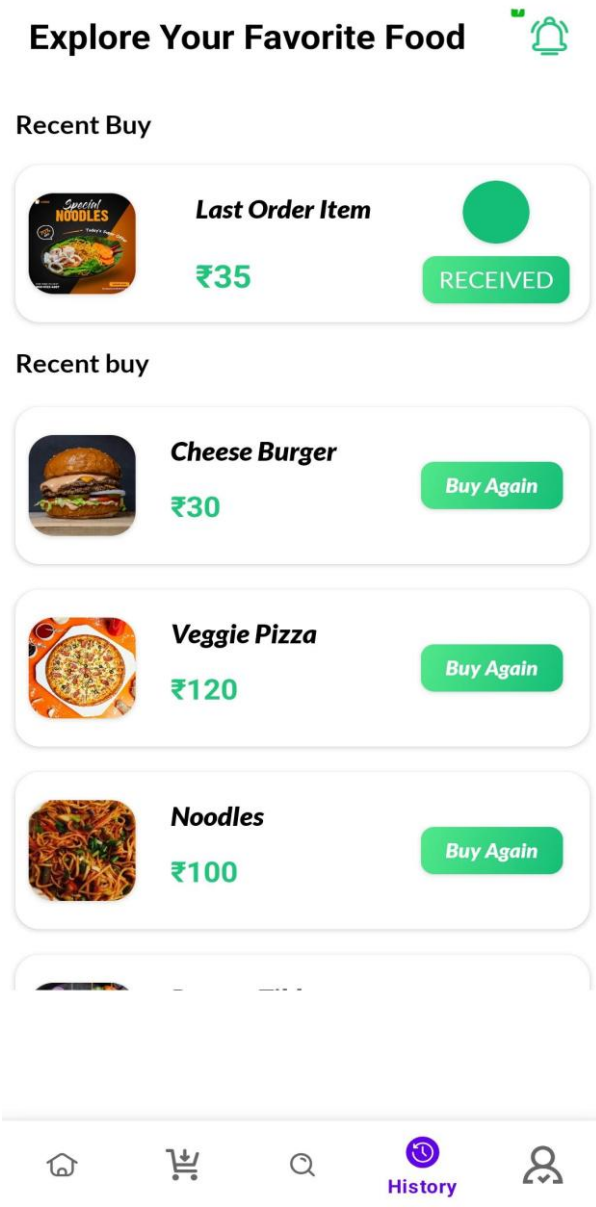
The **Cart Page** displays selected items with quantities and total price. Users can update or remove items before checkout.

6.7.2.6 Search Page



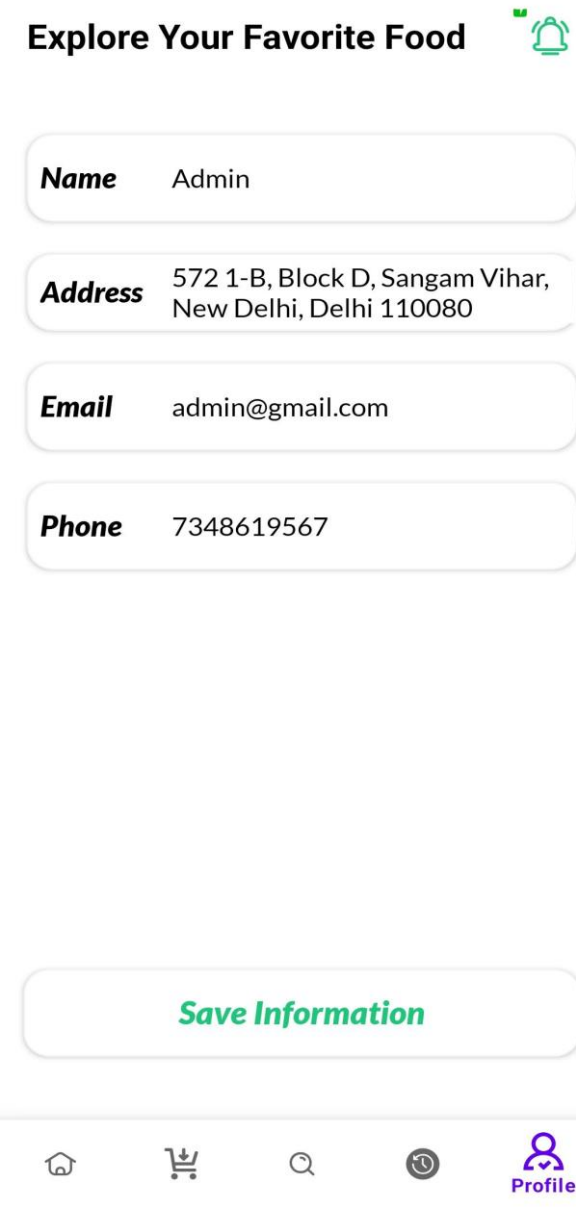
The **Search Page** allows users to find food items quickly. Results are shown with names, prices, and categories.

6.7.2.7 History Page



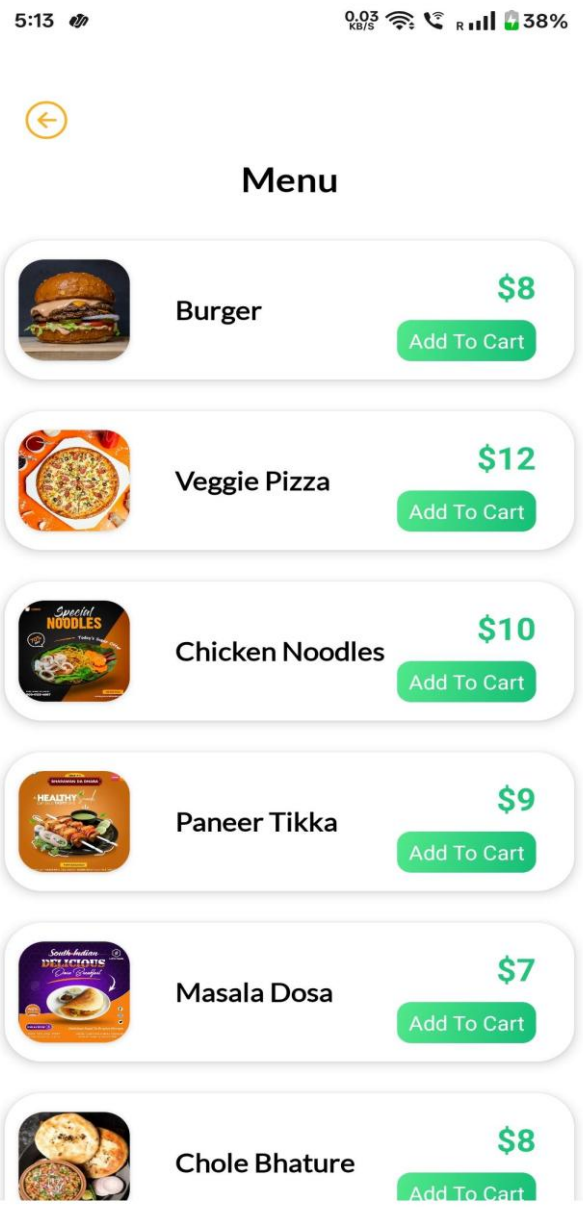
The **History Page** displays users' past orders with details like date, items, and total amount. It helps track previous purchases easily.

6.7.2.8 Profile Page



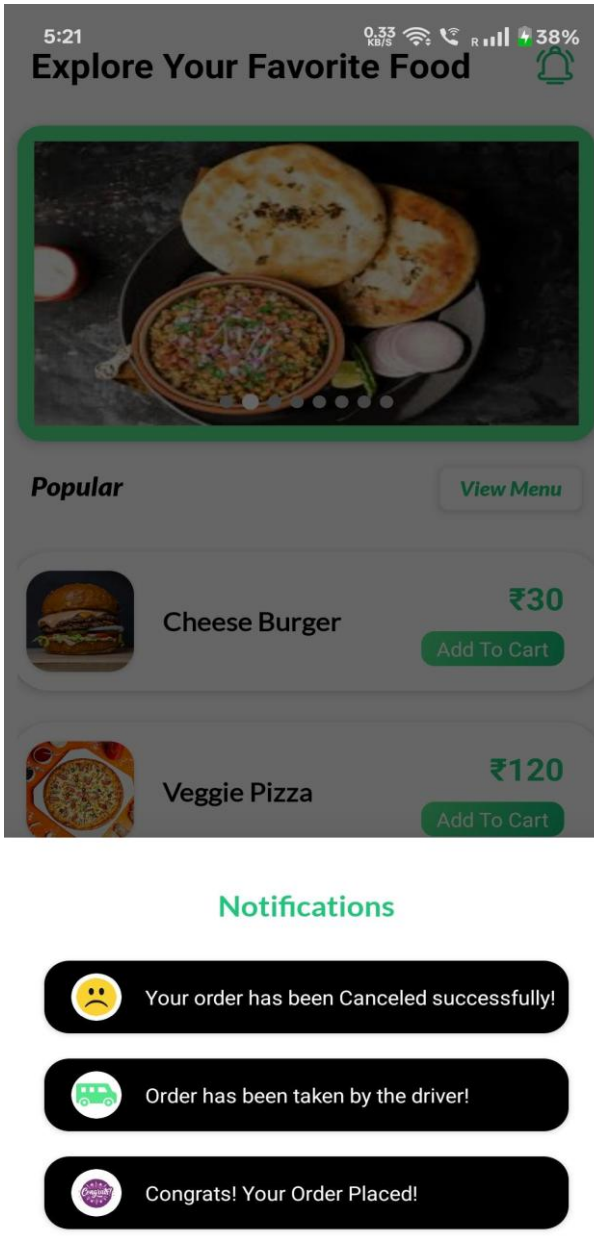
The **Profile Page** displays user details like name, email, and location. It also allows editing and logout options.

6.7.2.9 Menu Page



The **Menu Page** shows categorized food items with prices and images. Users can browse and add items to the cart easily.

6.7.2.10 Notification Page

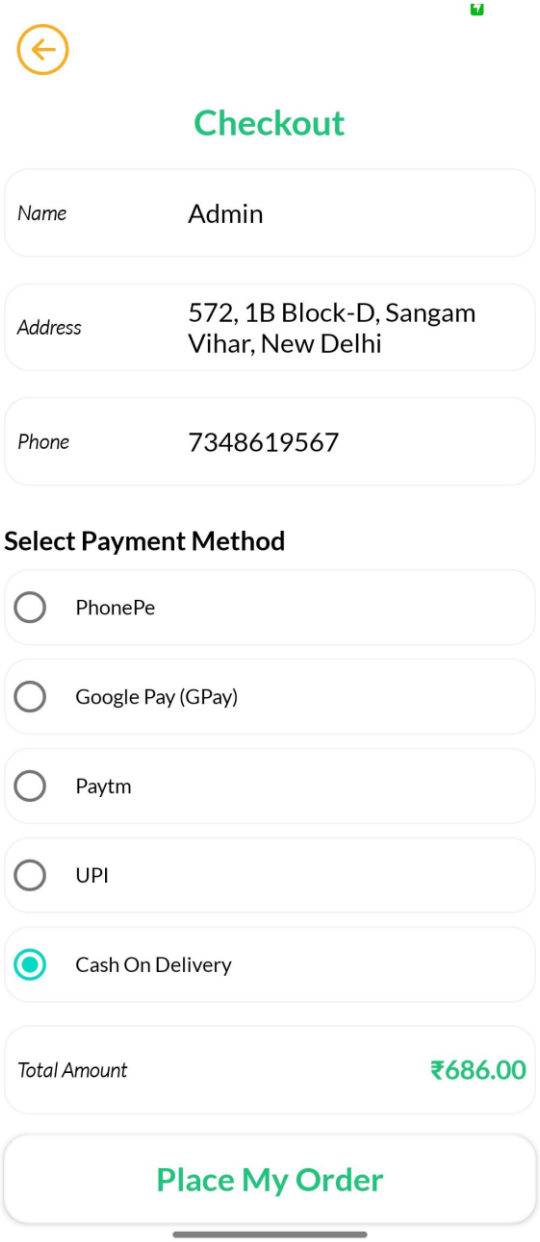


The **Notification Page** displays updates about orders, offers, and discounts. It keeps users informed in real time.

6.7.2.11 Details_Item Page

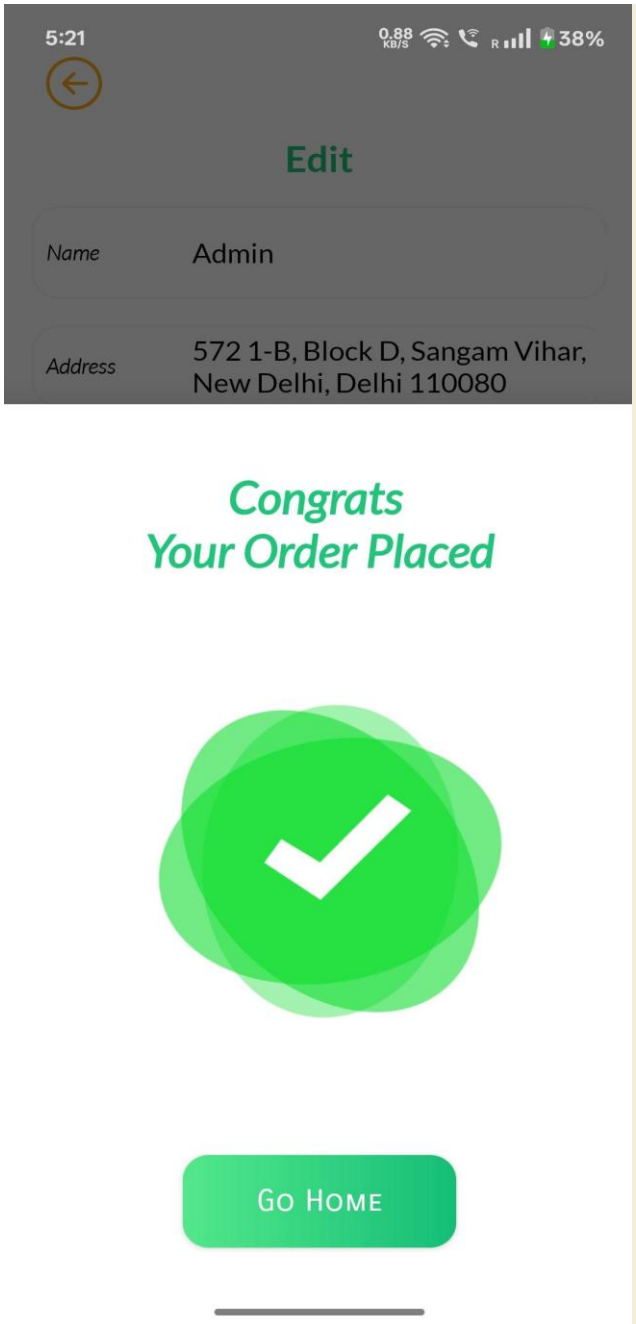
The **Detail Item Page** shows complete information about a selected food item, including image, description, and price. It allows users to add the item to their cart with customization options.

6.7.2.12 Payment Page



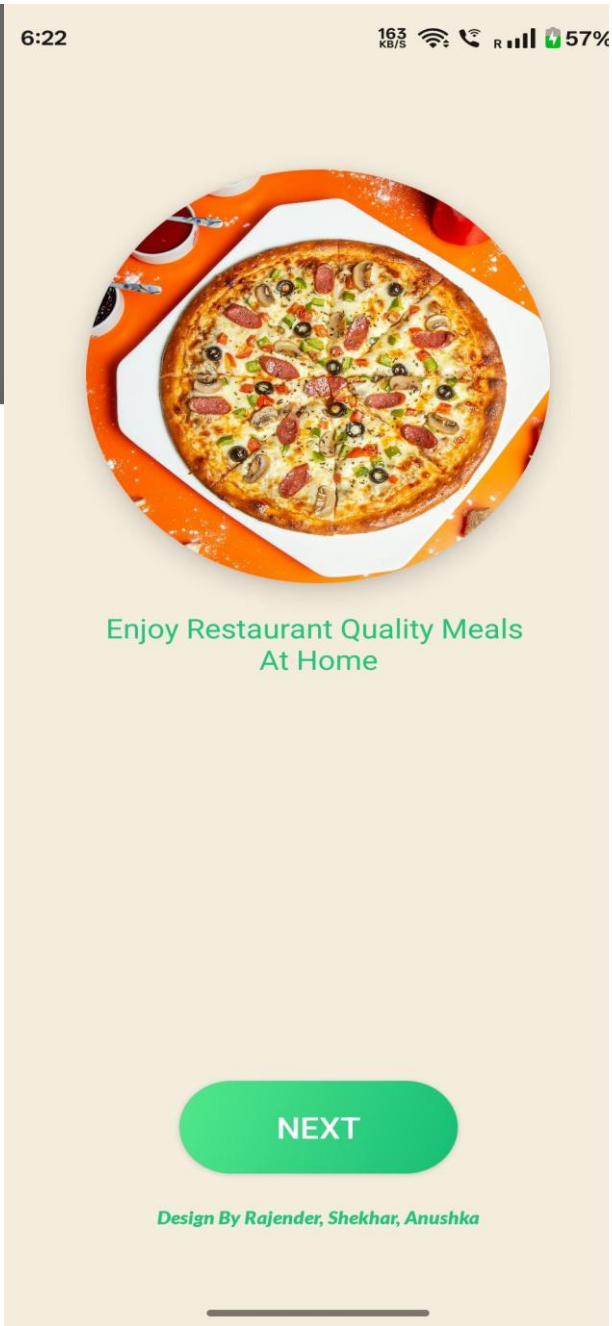
The **payment page** allows users to securely enter their payment details like card or UPI information. It provides options for transaction confirmation and displays success or failure messages.

6.7.2.13 Congrats Page



The Congrats page shows a success message after completing an action. It may include order details or next steps for the user.

6.7.2.14 Splash_Screen Page



The splash screen appears when the app launches, displaying the app logo or branding. It provides a brief loading experience before navigating to the main page.

Chapter 7: Future Scope, Conclusion, and References

7.1 Future Scope

The *OrderEats* application has been designed as a functional prototype to solve the fundamental problem of digital food ordering with clean UI, modular code, Firebase backend integration, and a streamlined ordering flow. While the system achieves its primary objectives, there is significant potential for expansion in terms of usability, scalability, features, and user satisfaction. This section presents the future scope of OrderEats, focusing on both technical enhancements and strategic improvements.

7.1.1 Online Payments Integration

Currently, the prototype relies on basic order confirmation without an in-app payment feature. A major future scope is the seamless integration of online payment gateways. The payment system should support:

UPI Integration: Leveraging Unified Payments Interface (UPI) for real-time, instant transactions.

Wallet Support: Compatibility with Google Pay, Paytm, PhonePe, and Amazon Pay.

Card Payments: Credit and debit card options with OTP verification.

Secure Callbacks: Ensuring end-to-end encryption and token-based authentication during transactions.

Refund & Cancellation Policies: Handling failed payments, partial refunds, and automatic wallet credits.

Secure payment integration will enhance trust among users and help restaurants expand their digital revenue channels.

7.1.2 Live Tracking of Deliveries

Another essential feature is real-time delivery tracking. Customers increasingly expect transparency about when their order will arrive. Using the Google Maps SDK and GPS APIs, the following can be achieved:

Delivery Partner Location Tracking: Displaying the rider’s current position.

Estimated Time of Arrival (ETA): Dynamically updating based on traffic and distance.

Route Optimization: Suggesting the shortest or fastest path for the delivery partner.

Privacy Features: Customers should see only approximate locations to protect rider privacy.

Live tracking enhances customer satisfaction by providing control and reducing anxiety regarding food delivery timelines.

7.1.3 Push Notifications, Offers, and Campaigns

To retain customers and increase engagement, personalized push notifications and discount campaigns can be integrated using Firebase Cloud Messaging (FCM).

Discount Alerts: Notifications for limited-time offers and coupons.

Order Status Updates: Informing customers when food is being prepared, picked up, and delivered.

Personalized Recommendations: Based on past order history, cuisine preferences, and festivals.

Reminders: Nudging inactive users to place orders with incentives like cashback.

Push campaigns improve customer loyalty and allow restaurants to compete effectively in a crowded marketplace.

7.1.4 Restaurant Dashboard (Partner Portal)

The *OrderEats* ecosystem can be expanded by developing a Restaurant Dashboard as a web portal or partner mobile app. This dashboard would allow restaurants to:

Manage their menu items, add/remove dishes, and update pricing.

Receive and track incoming orders in real time.

Update inventory to avoid overselling unavailable items.

Generate sales and revenue reports for business insights.

Respond to customer ratings and feedback.

Such a dashboard empowers restaurants to take control of their digital presence and ensures smoother coordination between users, delivery partners, and businesses.

7.1.5 AI-Powered Recommendation Engine

Personalization is the cornerstone of modern apps. An AI-driven recommendation engine could analyze user data to suggest items based on:

Order History: Suggest frequently ordered meals or favorites.

Contextual Data: Recommend warm beverages on rainy days or cold drinks in summer.

Collaborative Filtering: Suggesting popular dishes ordered by similar customer profiles.

Machine Learning Models: Using clustering or neural networks for smart menu ranking.

A well-designed recommendation system can increase average order value and enhance customer delight.

7.1.6 Accessibility Features

Future iterations should focus on inclusive design to make OrderEats accessible to differently-abled users. Some features include:

Screen Reader Support: Ensuring compatibility with TalkBack (Android) for visually impaired users.

Larger Fonts and High Contrast Modes: Supporting users with low vision.

Voice Search: Allowing users to search menus via speech.

Color-Blind Friendly Palettes: Using patterns/text instead of only colors to denote categories.

These features ensure compliance with WCAG (Web Content Accessibility Guidelines) and promote inclusivity.

7.1.7 Scalability and Cloud Deployment

As the app grows, scalability will be crucial. Potential future directions include:

Migration to Cloud Services: Hosting backend services on Google Cloud, AWS, or Azure.

Load Balancing: Handling peak-hour traffic with distributed servers.

Database Sharding: Splitting Firebase or SQL databases into smaller units for faster queries.

Microservices Architecture: Dividing the app into independent modules for easier maintenance.

Scalability planning ensures the app can handle thousands of users without performance issues.

7.1.8 Internationalization and Multi-Language Support

For global expansion, the app should support:

Multi-Language Interfaces: Hindi, English, and regional languages in India; global languages for international use.

Local Menus: Adapting restaurant listings and food items to the cultural preferences of each region.

Internationalization will transform *OrderEats* from a local app into a globally competitive platform.

7.2 Conclusion

The *OrderEats* project represents the journey of conceptualizing, designing, developing, and testing a food ordering application within a limited timeframe. The app demonstrates that even with a small, focused team, it is possible to create a product that is both functional and scalable.

Key achievements include:

Clean User Interface: Simple XML-based designs ensure easy navigation.

Modular Codebase: Implemented in Java with well-defined adapters, fragments, and activities.

Firebase Integration: For authentication, database management, and notifications.

System Planning: PERT charts and timelines were used for efficient scheduling.

Testing Methodologies: Unit, integration, and acceptance testing ensured reliability.

7.2.1 Learning Outcomes

Working on *OrderEats* helped the team develop several technical and non-technical skills:

Technical Learning: Android fundamentals, Firebase, API handling, XML layouts, and testing frameworks.

Collaboration: Improved teamwork, task distribution, and communication within the group.

Problem-Solving: Overcoming challenges such as Firebase configuration, RecyclerView adapters, and authentication bugs.

Documentation: Creating detailed project reports improved academic and professional writing.

7.2.2 Industry Relevance

In the real-world context, online food ordering has become a multi-billion-dollar industry with apps like Swiggy, Zomato, and Uber Eats. *OrderEats* is aligned with industry trends and serves as a training ground for entering professional software development.

While not as large as commercial systems, it captures the core workflow: browsing restaurants, selecting menu items, placing orders, processing data, and preparing for delivery.

7.2.3 Limitations and Improvements

Some limitations of the current prototype include:

Lack of real payment integration (currently simulated).

Absence of AI recommendations and tracking features.

Limited partner tools for restaurants and delivery agents.

These limitations form the basis of the future scope discussed earlier. With additional development, *OrderEats* can evolve into a competitive market-ready solution.

wo

7.3 References

1. Android Developers. *Android Development Documentation*. Retrieved from: https://developer.android.com
2. Firebase Documentation. *Backend-as-a-Service Features*. Retrieved from: https://firebase.google.com/docs
3. Google Material Design Guidelines. *UI/UX Standards for Android*. Retrieved from: https://material.io
4. Google Maps Platform. *Maps SDK for Android*. Retrieved from: https://developers.google.com/maps/documentation
5. Jetpack Navigation Component. *Managing Navigation in Android*. Retrieved from: https://developer.android.com/guide/navigation
6. Pressman, R. S. (2014). *Software Engineering: A Practitioner's Approach*. McGraw-Hill Education.
7. Sommerville, I. (2015). *Software Engineering* (10th ed.). Pearson Education.
8. Nielsen, J. (1993). *Usability Engineering*. Academic Press.
9. ISO/IEC 25010:2011. *Systems and software engineering – Quality models*. International Standards Organization.
10. Stack Overflow Discussions. *Community Insights on Android Development*.