

MINI PROJECT REPORT

on

ORDERBYTE: A SMART CANTEEN WEBSITE

Submitted by

MERLYN MARY STEPHEN MGP22UCS093

RINNY ANNA THOMAS MGP22UCS111

SHALIN ANN THOMAS MGP22UCS123

SNEHA MARIAM SAMUEL MGP22UCS131

To APJ Abdul Kalam Technological University in partial fulfilment of the requirements for
the award of the degree of

Bachelor of Technology

in

Computer Science & Engineering



Department of Computer Science and Engineering

Saintgits College of Engineering (Autonomous)

APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

April 2025

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
SAINTGITS COLLEGE OF ENGINEERING (Autonomous)



2024-2025

CERTIFICATE

Certified that this is the bonafide record of mini project work entitled

ORDERBYTE: A SMART CANTEEN WEBSITE

Submitted by

MERLYN MARY STEPHEN MGP22UCS093

RINNY ANNA THOMAS MGP22UCS111

SHALIN ANN THOMAS MGP22UCS123

SNEHA MARIAM SAMUEL MGP22UCS131

Under the guidance

of

Er. Thomas Joseph

*In partial fulfilment of the requirements for award of the degree of Bachelor of Technology in
Computer Science and Engineering under the APJ Abdul Kalam Technological University during
the year 2024-2025.*

HEAD OF DEPARTMENT

PROJECT COORDINATOR

PROJECT GUIDE

Dr. Arun Madhu

Er. Prince V Jose

Er. Thomas Joseph

DECLARATION

We, the undersigned hereby declare that the project report 'ORDERBYTE: A SMART CANTEEN WEBSITE', submitted for partial fulfilment of the requirements for the award of degree of Bachelor of Technology of the APJ Abdul Kalam Technological University, Kerala is a bonafide work done by us under supervision of Er. Thomas Joseph. This submission represents our ideas in our own words and where ideas or words of others have been included, we have adequately and accurately cited and referenced the original sources. We also declare that we have adhered to ethics of academic honesty and integrity and have not misrepresented or fabricated any data or idea or fact or source in my submission. We understand that any violation of the above will be a cause for disciplinary action by the institute and/or the University and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been obtained. This report has not been previously formed the basis for the award of any degree, diploma or similar title of any other University.

Place

Signature

Date

Merlyn Mary Stephen
Rinny Anna Thomas
Shalin Ann Thomas
Sneha Mariam Samuel

ACKNOWLEDGEMENT

We express our gratitude to **Dr. Sudha T**, Principal, Saintgits College of Engineering for providing us with excellent ambiance that laid potentially strong foundation for this work.

We express our heartfelt thanks to **Dr. Arun Madhu**, Head of the Department of Computer Science and Engineering, Saintgits College of Engineering who has been a constant support in every step of our seminar and the source of strength in completing this mini project.

We express our sincere thanks to **Er. Thomas Joseph**, Computer Science and Engineering Department for providing us with all the facilities, valuable and timely suggestions and constant supervision for the successful completion of our mini project.

We are highly indebted to project coordinator, **Er. Prince V Jose** and all the other faculties of the department for their valuable guidance and instant help and for being with us. We extend my heartfelt thanks to our parents, friends and well-wishers for their support and timely help.

Last but not the least we thank Almighty God for helping us in successfully completing this mini project.

ABSTRACT

Topic: OrderByte: A Smart Canteen Website

OrderByte is an advanced canteen management system developed to enhance food service operations within educational institutions. It offers a wide range of features aimed at improving efficiency, reducing food wastage, and ensuring a seamless ordering experience for users. The system supports secure user registration, password recovery, and profile customization, including updating contact details and profile images.

Users can explore categorized menus with daily specials, apply search filters to find specific items, and manage their selections using a digital cart. Integrated third-party payment processing ensures secure and smooth transactions, with each order generating a unique ID and invoice. Order history and cancellation options further add to user convenience. A built-in complaint system allows users to report issues directly to the administration for timely resolution.

For administrators, OrderByte provides a dedicated control panel that allows menu management, including adding, editing, deleting, and publishing food items. Features such as stock tracking, marking items as out of stock, and applying discounts help in real-time inventory control. Admins can also view and manage daily orders and monitor user complaints. Profile customization options are available for administrators as well, ensuring a personalized experience.

With its emphasis on automation, real-time updates, and comprehensive management tools, OrderByte simplifies daily canteen operations, increases staff productivity, and enhances customer satisfaction in both small and large institutional settings.

TABLE OF CONTENTS

INDEX	PAGE NO
ABSTRACT -----	V
LIST OF FIGURES -----	VIII
1. INTRODUCTION -----	1
2. LITERATURE REVIEW -----	3
3. REQUIREMENT ANALYSIS -----	5
3.1 Feasibility Study -----	5
3.2 Software Requirement -----	5
3.2.1 Python Programming Language -----	5
3.2.2 Firebase Database Software-----	5
3.2.3 Hypertext Markup Language (HTML) -----	6
3.2.4 JavaScript Programming Language -----	6
3.2.5 Cascading Style Sheets (CSS) -----	6
3.2.6 RazorPay (Third-Party Payment Integration) -----	6
3.3 Hardware Requirement-----	7
3.3.1 Web Hosting or Server/Local hosting and Deployment-----	7
3.3.2 Database Server-----	7
3.3.3 Internet Connection-----	7
3.3.4 Computer or Development Systems-----	7
3.3.5 Applications -----	8
3.3.6 Advantages -----	8
3.3.7 Disadvantages -----	8
3.4 General Requirement -----	9
3.4.1 Accessing OrderByte Website -----	9
3.4.2 Interacting with OrderByte -----	9
3.4.3 Placing and Managing Orders -----	9
3.4.4 Administrative Functions-----	9
4. DESIGN -----	11
4.1 Use Case Diagram-----	11
4.2 Workflow Diagram-----	12
4.3 Features -----	13
4.3.1 User Features -----	13
4.3.2 Admin Features-----	13
4.4 Architecture & Technology-----	13

4.4.1 System Architecture -----	13
4.4.2 Technology Stack -----	14
4.5 Management -----	14
4.5.1 User Management-----	14
4.5.2 Admin Management -----	14
4.6 User Interface -----	15
4.6.1 User Authentication-----	15
4.6.2 Home Page -----	15
4.6.3 Search and Filter -----	15
4.6.4 Cart and Checkout-----	16
4.6.5 Profile Page -----	16
4.6.6 Complaint and Feedback-----	16
4.6.7 Logout -----	16
4.7 Admin Interface -----	16
4.7.1 Admin Authentication -----	16
4.7.2 Menu Management -----	17
4.7.3 Order Management -----	17
4.7.4 Complaint Management-----	17
4.7.5 Profile Management -----	17
4.7 System Flow -----	17
5. DEVELOPMENT -----	19
5.1 Requirement Analysis -----	19
5.2 Technology Selection-----	20
5.3 Design -----	21
5.4 Development-----	21
5.5 Testing -----	22
5.6 Deployment -----	22
5.7 Technology Stack -----	23
5.8 System Implementation-----	23
5.9 Error Handling Mechanism -----	23
5.10 Monitoring and Maintenance-----	24
6. TESTING & MAINTANENCE -----	25
6.1 Testing Process -----	25
6.1.1 Functional Testing-----	25
6.1.2 Performance Testing -----	25
6.1.3 Security Testing -----	26
6.2 Testing Results -----	26
6.1.1 Functional Issues -----	26
6.1.2 Performance Issues-----	26
6.1.3 Interface Issues -----	27
6.3 Maintenance -----	27

7. CONCLUSION	-----28
8. REFERENCES	-----30
9. APPENDIX	-----31

LIST OF FIGURES

FIGURE NO.	TITLE	PAGE NO.
4.1	OrderByte Use Case Diagram	11
4.2	OrderByte Workflow Diagram	12
4.3	OrderByte User Profile Interface	14
4.4	OrderByte User Profile Order History	15
4.5	OrderByte Admin Interface	15
5.1	Sign-In and Sign-Up Pages of OrderByte	21

CHAPTER 1

INTRODUCTION

Canteen systems in many educational institutions continue to face recurring issues such as long queues, inefficient order processing, food wastage, and service delays. These problems often lead to dissatisfaction among students, faculty, and visitors, diminishing their overall dining experience. To resolve these concerns, the integration of a smart, automated system is essential for boosting service speed, minimizing waste, and optimizing resource usage.

The OrderByte: A Smart Canteen Website is a modern digital solution aimed at upgrading food service operations across academic campuses. It addresses the problems of traditional setups by offering categorized menus, meal pre-booking, secure online payments, and real-time inventory tracking—ensuring a fast, organized, and reliable canteen experience for both users and staff.

Menus are clearly organized into categories such as breakfast, lunch, drinks, curry and desserts, making food selection simple and efficient. Users can view food items, add them to their cart, and complete transactions smoothly. For added convenience, the system allows advance meal reservations, catering to those with busy routines.

Administrators benefit from a real-time dashboard that enables them to manage menus, track orders, monitor inventory, and update item status or apply discounts as needed. The system also supports user profile customization, allowing individuals to update personal information and preferences.

This project ultimately seeks to revolutionize traditional canteen operations by delivering a smarter, more efficient, and user-focused solution tailored to the needs of modern educational institution. Beyond convenience, it also supports sustainable practices by reducing food waste and improving operational flow.

1.1 Project Objective

The main goal of OrderByte: A Smart Canteen Website is to digitize and simplify the food ordering process in educational environments. It aims to minimize waiting times by providing an online platform accessible to students, faculty, and campus visitors. The system organizes food items into categories like breakfast, lunch, drinks, desserts, and daily specials to make browsing easier. It supports meal pre-booking to help users avoid delays, while secure digital payment options enable a smooth, cashless experience. Additional features include user profile customization and access

to past orders for improved convenience. For administrators, the system offers real-time order monitoring, menu management, and inventory tracking, helping to reduce food waste and boost overall efficiency.

1.2 Project Scope

The scope of this project involves building a robust web-based application specifically designed to enhance college canteen operations. It enables digital food ordering and meal pre-booking, allowing users to place and track orders without waiting in long lines. For administrators, the system offers a centralized dashboard to manage menu items, monitor ongoing orders, and oversee inventory levels in real time. Secure payment integration ensures safe and cashless transactions, increasing user convenience. A built-in complaint management feature also allows users to share feedback or raise issues. Developed with scalability in focus, OrderByte can be implemented across different educational institutions and helps minimize food wastage through efficient demand prediction.

1.3 Project Overview

The OrderByte: A Smart Canteen Website is a modern solution designed to upgrade the traditional canteen setup into a more efficient, digital experience. It allows users to explore organized menu categories, search for food items, add them to their cart, and complete secure online payments. With the option to pre-book meals, users can avoid peak-time delays and ensure timely service. The platform also offers a personalized touch through user profiles, enabling individuals to track past orders and update personal details. From the administrative perspective, the system provides tools to manage food items, track orders, oversee inventory, and address user complaints through a unified dashboard. Ultimately, OrderByte improves operational efficiency, cuts down on food wastage, and delivers a smoother, more convenient dining experience for everyone involved.

CHAPTER 2

LITERATURE REVIEW

Technology has transformed food service operations, particularly in canteen management, by introducing automation, digital payments, and real-time tracking systems. These advancements have improved efficiency, minimized waiting times, and enhanced customer experience. This section reviews key studies that explore the implementation of digital canteen management systems, their advantages, and the challenges they address.

Automated systems have streamlined traditional canteen processes by simplifying food selection, billing, and inventory management. Lalitha, V., et al. [1] discuss how a web-based canteen system enhances service efficiency by reducing manual workload and improving order processing. Their research highlights how digital ordering platforms help optimize canteen operations, resulting in faster service and a more organized workflow.

Long queues and congestion are common issues in canteens, leading to inefficiencies and customer frustration. Xu, Zixuan, et al. [2] introduce a queue-free digital ordering system designed to allow users to pre-order meals through an interactive menu. This approach ensures timely meal preparation while also promoting healthier food choices by integrating nutritional information. By leveraging digital technology, the study demonstrates how canteens can reduce service delays and enhance overall dining experiences.

Managing peak-hour congestion remains a challenge for many food service establishments. To address this, Ardiansyah, Muhammad Rizqi, et al. [3] explore the benefits of real-time order tracking in digital food booking systems. Their research shows that live updates on order status provide greater transparency, improve kitchen efficiency, and reduce uncertainty for customers. Implementing real-time tracking ensures a smoother service flow and enhances customer trust in digital ordering platforms.

The shift to digital transactions has revolutionized the payment process in food services, making transactions faster and more secure. Avhad, Prashant, et al. [4] discuss the integration of cashless payment solutions within canteens, highlighting their ability to reduce human errors and improve financial management. Their study emphasizes that digital payment systems not only offer convenience but also contribute to better financial transparency and record-keeping in canteen operations.

The integration of IoT (Internet of Things) and RFID (Radio Frequency Identification) technology has introduced new possibilities in canteen automation. Gosalia, Jay, et al. [5] explored an advanced canteen management system incorporating RFID-based payment solutions and automated table reservations. Their research demonstrates how IoT-enabled systems can enhance order accuracy, reduce service delays, and provide a more personalized dining experience. Additionally, the study highlights how AI-driven recommendations can further optimize food choices based on user preferences.

As canteens continue to embrace digital transformation, ongoing system maintenance and updates remain essential to ensure smooth operations. Future developments may include AI-powered chatbots for automated order assistance, further enhancing the user experience. Additionally, security measures must be strengthened to protect sensitive user data, ensuring compliance with digital privacy regulations.

The continuous evolution of smart canteen systems highlights the importance of automation in modern food services. By adopting innovative digital solutions, canteens and restaurants can provide faster, more efficient, and user-friendly services, setting new standards for industry.

CHAPTER 3

REQUIREMENT ANALYSIS

3.1 Feasibility Study

Traditional canteen systems in educational institutions frequently face challenges such as long queues, excessive food waste, and inefficient order processing. OrderByte offers a modern solution that streamlines the food ordering process and optimizes operations through digital automation. By using Firebase for real-time database updates and Flask for its backend, OrderByte ensures effective data management and timely information flow. Its intuitive interface and live features enhance the dining experience for students, faculty, and visitors by reducing wait times, optimizing resource utilization, and maintaining affordability without compromising service quality. Moreover, the solution includes robust analytics tools that empower administrators to monitor usage trends and refine inventory management. Its flexible, scalable architecture allows the system to grow alongside the institution, meeting increasing demand and adapting to evolving needs over time.

3.2 Software Requirement

3.2.1 Python Programming Language

Python is a robust and versatile programming language widely utilized for backend development. Its clean syntax and extensive library support streamline the development process, while its integration with Firebase enables real-time data synchronization for functionalities such as menu updates, order tracking, and inventory control. Python Flask is a minimalist web framework designed for Python, ideal for constructing small to medium-sized web applications. Flask is easy to learn and extend, making it perfect for rapid development and efficient feature integration.

3.2.2 Firebase Database Software

Firebase is a cloud-based database platform that delivers real-time synchronization, robust authentication, and scalable NoSQL storage solutions. It facilitates instantaneous data updates, controlled access, and smooth integration with both web and mobile applications, making it well-suited for interactive, data-centric systems. Its built-in security features and serverless architecture simplify backend operations while maintaining high efficiency. Firebase provides an extensive range of development tools and analytics that streamline and enable rapid adaptation to evolving application requirements.

3.2.3 Hypertext Markup Language (HTML)

HTML (HyperText Markup Language) serves as the backbone for building and structuring content on the web. It organizes elements such as text, images, links, and forms to create meaningful page layouts. When used in combination with CSS and JavaScript, it enables the development of interactive and visually appealing websites. HTML's standardized syntax ensures consistent behavior across different browsers and platforms. It also supports semantic tagging, improving accessibility and search engine visibility.

3.2.4 JavaScript Programming Language

JavaScript is a versatile scripting language used to introduce interactivity and dynamic content on websites. It enables features such as form validation, animations, and live updates, significantly enhancing user experience. Operating within the browser, it works seamlessly with HTML and CSS to build responsive, real-time applications. JavaScript also supports a wide range of libraries and frameworks, enabling scalable and feature-rich development. Its adaptability and extensive ecosystem make it essential for building modern, interactive web solutions.

3.2.5 Cascading Style Sheets (CSS)

CSS (Cascading Style Sheets) is a design language used to style and format the layout of web content. It complements HTML by managing visual aspects such as color schemes, typography, spacing, and element positioning. CSS supports responsive design, allowing websites to adapt seamlessly across various devices and screen sizes. By separating content from presentation, it simplifies website maintenance and customization. Its versatility enables developers to build clean, engaging, and accessible user interfaces. Additionally, CSS frameworks like Bootstrap speed up development by offering pre-designed components and layouts.

3.2.6 RazorPay (Third-Party Payment Integration)

Razorpay is a reliable payment gateway that supports various digital payment methods such as Unified Payments Interface (UPI), cards, net banking, and wallets. It allows easy and secure integration into the web and mobile platforms, enabling quick and efficient online transactions. With real-time status tracking, user-friendly APIs, and built-in security measures, Razorpay streamlines the payment process while safeguarding user data. Its flexible and developer-focused design helps improve the overall experience for both users and service providers in digital payment systems.

3.3 Hardware Requirement

3.3.1 Web Hosting or Server/Local hosting and Deployment

To ensure the system remains constantly available and delivers smooth user interactions, its code should be deployed on a dependable hosting platform. Various hosting options exist, including cloud services such as AWS and Google Cloud, as well as dedicated servers that offer the necessary resources to manage anticipated user traffic.

Alternatively, the system can be hosted locally on a single machine that operates 24/7. While this approach is more budget-friendly, it tends to be less reliable and provides fewer features compared to cloud-based solutions.

3.3.2 Database Server

A real-time database management system may be necessary to store and access essential data efficiently. A dedicated database server can help manage storage and retrieval while maintaining optimal performance and scalability. We utilize Firebase as the database solution, which provides real-time data synchronization, reducing the need for a separate database server. However, if local servers are preferred, a cloud-based solution or local database like **MySQL** or **PostgreSQL** could also be deployed, depending on the scale and customization required.

3.3.3 Internet Connection

A reliable internet connection is vital for the system's development and deployment. It facilitates smooth database communication, supports real-time updates, and ensures continuous accessibility for users. The connection must support high traffic and ensure uninterrupted service, especially during peak hours when many students, faculty, and staff place orders simultaneously.

3.3.4 Computer or Development Systems

A computer or laptop is required for development and testing. It must fulfill the minimum hardware and software specifications of the selected development environment and programming language. The device should have sufficient performance and compatibility to support both the creation and deployment phases effectively.

- CPU: Dual-core or higher (Intel i5/Ryzen 5 or equivalent)
- RAM: 8 GB or more
- Storage: SSD with at least 100 GB of available space

- Network: Stable internet connection for interacting with Firebase, testing the web application, and using third-party APIs such as payment gateways

For local deployment, a server with at least the following specifications may be required:

- CPU: Quad-core or higher (Intel i7 / Ryzen 7 or equivalent)
- RAM: 16 GB or higher
- Storage: SSD (500 GB or more)
- Network: High-speed internet for hosting and serving user requests

3.3.5 Applications

1. Educational Institutions: OrderByte transforms canteen operations in educational institutions by digitalizing the food ordering process, delivering real-time menu updates, and streamlining order and inventory management.
2. Corporate Canteens: OrderByte optimizes corporate canteens by providing cheaper meals, real-time menu updates, streamlining processes, reducing waste, and boosting employee satisfaction.
3. School Cafeterias: OrderByte can help manage lunch schedules by allowing students or parents to pre-order meals. This ensures timely service, better nutrition tracking, and reduced food wastage through precise demand forecasting.

3.3.6 Advantages

1. Time Efficiency: Minimizes wait times by leveraging online ordering and pre-booking options.
2. Convenience: Offers an intuitive interface that makes ordering and payment quick and hassle-free.
3. Affordability: Delivers budget-friendly meal options that are more cost-effective than those offered by external vendors.
4. Resource Optimization: Reduces food waste through real-time inventory tracking.
5. Feedback System: Facilitates direct communication by enabling users to submit complaints to administrators for prompt resolution.

3.3.7 Disadvantages

1. Internet Dependency: System performance is reliant on stable connectivity.
2. Technical Learning Curve: Staff may need training to use the admin panel efficiently.

3. Maintenance Needs: Regular updates and security measures must be implemented to ensure smooth operation.
4. Limited Payment Security in Test Mode: During development, Razorpay's test mode simulates secure payments without real-world banking or OTP verification. Final deployment requires switching to live mode for full payment security compliance.

3.4 General Requirements for OrderByte

3.4.1 Accessing OrderByte Website

- Users should be able to easily access OrderByte through any standard web browser or mobile application on their device using email and password.
- If a user forgets their password, they have the option to reset it quickly.
- The platform must load swiftly and consistently, providing a seamless user experience across various devices.
- All expected functionalities should be fully operational and available upon access.

3.4.2 Interacting with OrderByte

- Users must be able to navigate the digital menu effortlessly and view real-time updates.
- The system should allow users to select, adjust, and finalize their orders using clearly labelled navigation buttons and options.
- Smooth transitions between sections such as different menu categories, order summaries, and pre-booking interfaces are essential.

3.4.3 Placing and Managing Orders

- Users must be capable of adding items to a virtual cart, reviewing their choices, and completing the ordering process via the interface.
- The platform should support order pre-booking, catering to users with busy schedules and ensuring timely service.
- Real-time order tracking is required so that users can monitor their order status from placement through fulfilment.

3.4.4 Administrative Functions

- Administrators should have access to a dedicated dashboard for updating menus, managing inventory, and tracking orders in real time.

- The admin interface must offer intuitive controls and live insights to facilitate prompt operational adjustments.
- Administrators can set the daily menu and update it as needed, ensuring that the offerings remain current and aligned with demand.

CHAPTER 4

DESIGN

This chapter offers an in-depth overview of the design and architecture of OrderByte, a comprehensive website created to optimize the canteen ordering process. The platform allows users to place food orders online, complete digital payments, and monitor order statuses in real time, while providing canteen administrators with the tools to manage order queues, update menus, and maintain transaction records through a dedicated dashboard.

The following sections outline the key features, system architecture, order management workflows, administrative functionalities, and user interface design that together ensure a seamless and efficient experience for all users.

4.1 Use Case Diagram

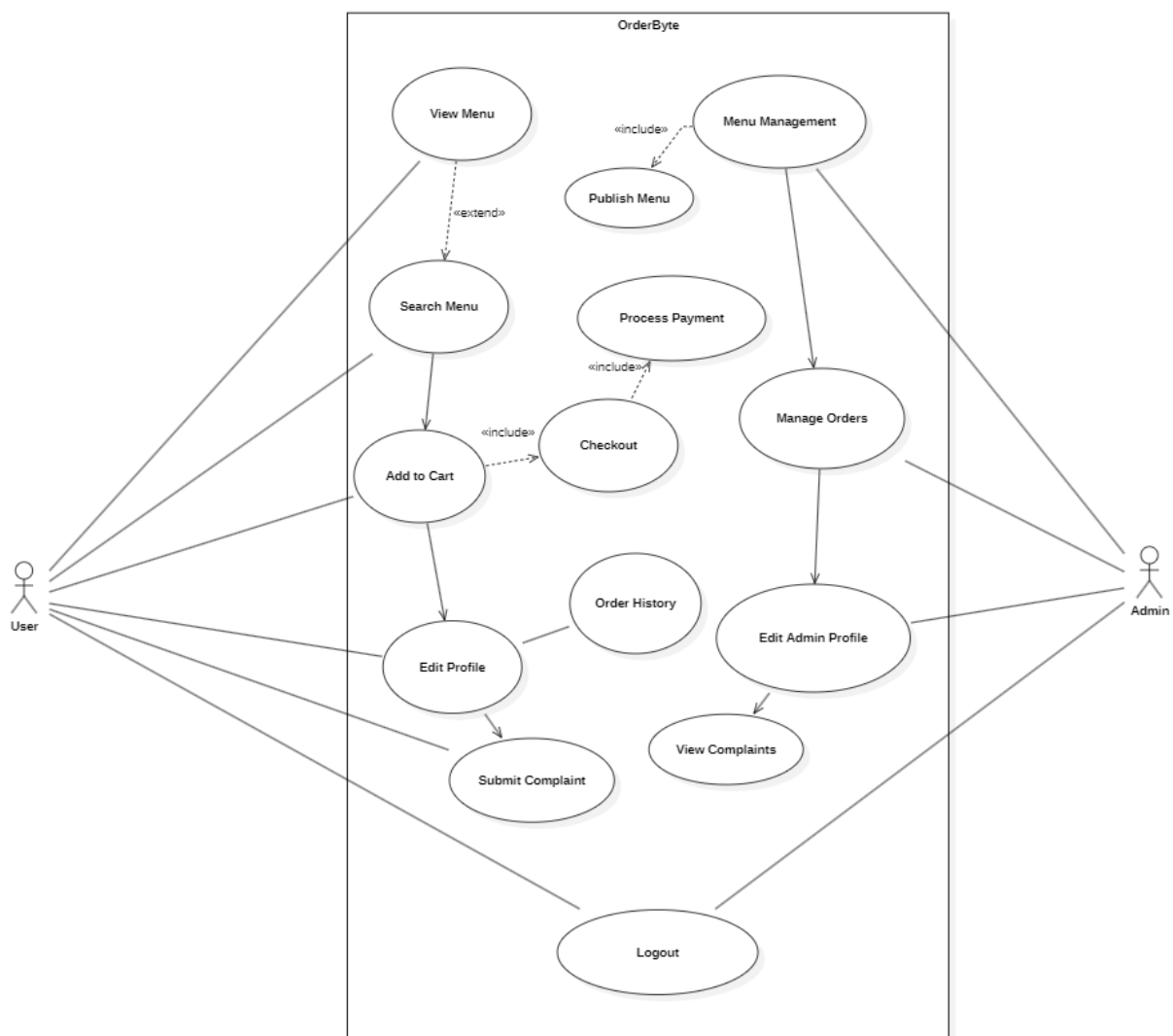


Figure 4.1: OrderByte use case diagram

The above use case diagram shows the structure of Orderbyte: A Smart Canteen Website. This system involves two main actors: Users (students, faculty, parents, and visitors) and Admins. Users can browse the menu, search items, add to cart, make payments, view order history, manage profiles, submit complaints, and log out. Admins manage menus, publish updates, process orders, respond to complaints, edit profiles, and handle system logout.

4.2 Workflow Diagram

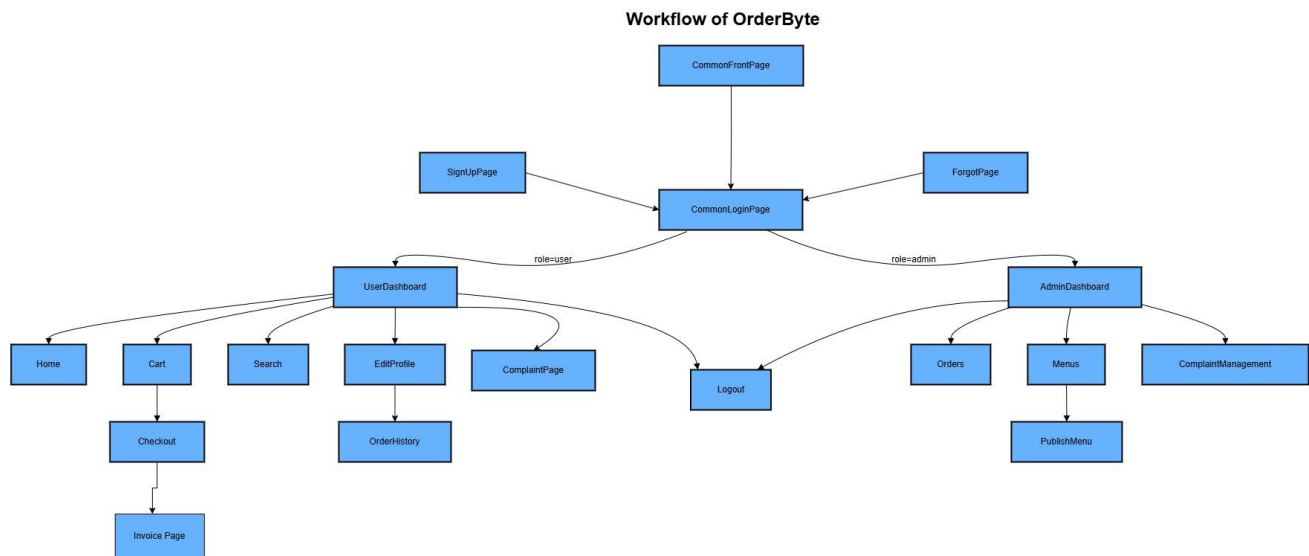


Figure 4.2: OrderByte workflow diagram

The workflow diagram showcases the navigational structure of the OrderByte canteen system. Users are first directed to the CommonFrontPage, from where they proceed to the CommonLoginPage for signing in. New users can register through the SignUpPage, while the ForgotPage allows password recovery for those who need it.

After logging in:

- Users are routed to the UserDashboard, where they can browse the Home page, search for food items, view their Cart, proceed to Checkout, and receive an Invoice Page. They can also manage personal details via the EditProfile section, review past activity through OrderHistory, file complaints through the ComplaintPage, and securely log out using the Logout function.
- Admins gain access to the AdminDashboard, enabling them to oversee Orders, update and release new menus through Menus and PublishMenu, address customer feedback under ComplaintManagement, and exit the session via the Logout option.

This role-based flow ensures that both users and administrators have access to the tools they need for efficient ordering and canteen management.

4.3 Features

4.3.1 User Features

- Menu Browsing: Categorized food items (Breakfast, Lunch, Drinks, Desserts, Curry)
- Search & Filter: Find food items by name, category, or price
- Cart Management: Add, modify, or remove items
- Profile Management: Edit personal details including profile picture, phone number, and email
- Instant Payment Processing: Quick and secure transactions through integrated third-party payment gateways
- Order Confirmation: Users receive a confirmation screen after completing the order
- Complaint Submission: Users can raise complaints or give feedback directly to the admin for resolution

4.3.2 Admin Features

- Menu Management: Add, edit, or remove food items
- Order Processing: View and manually mark orders as completed
- Inventory Control: Toggle item availability (Active/Inactive)
- Complaint Management: View and address user complaints efficiently to improve user satisfaction
- Profile Management: Update admin profile details including name, phone number, and profile picture
- Data Accuracy: Ensure users always view up-to-date menu options and availability with real-time synchronization

4.4 Architecture & Technology

4.4.1 System Architecture

The system follows a client-server architecture:

- Frontend: HTML, CSS, JavaScript for the user and admin interfaces
- Backend: Python Flask for handling requests and database interactions
- Database: Firebase for storing menu items, orders, and user data

4.4.2 Technology Stack

- Frontend: HTML, CSS, JavaScript
- Backend: Python Flask
- Database: Firebase
- Payment: RazorPay

4.5 Management

4.5.1 User Management

- Browse categorized menus, search/filter items, and manage cart contents.
- Place orders with secure third-party payment and receive confirmation.
- Track, view, or cancel orders from order history.
- Update personal profile details (phone, email, profile picture).
- Submit complaints to admin and feedback.

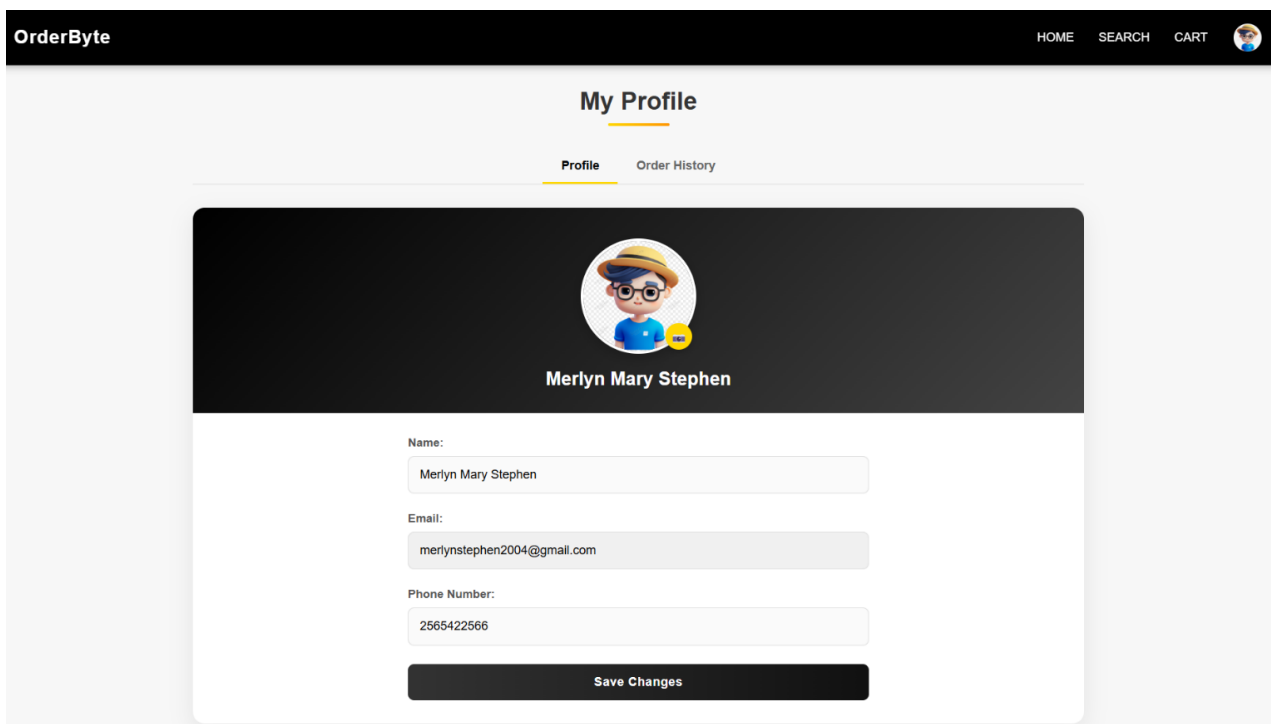



Figure 4.3: OrderByte user profile interface

OrderByte

HOME

SEARCH

CART



My Profile

Profile

Order History

MY ORDERS















DATE	ORDER ID	ITEMS	AMOUNT	STATUS	ACTIONS
2025-04-20 16:08:05	#order_QL	1 Item	₹70.0	CANCELLED	 
2025-04-20 14:12:00	#order_QL	2 Items	₹128.0	PAID	 
2025-04-19 23:56:29	#order_QL	2 Items	₹200.0	PAID	 
2025-04-19 23:53:31	#order_QL	1 Item	₹11660.0	PAID	 
2025-04-19 22:29:53	#order_QK	1 Item	₹40.0	PAID	 
2025-04-19 19:15:05	#order_QK	1 Item	₹120.0	CANCELLED	 
2025-04-19 18:28:39	#order_QK	1 Item	₹1.0	PAID	 

Figure 4.4: OrderByte user profile order history

4.5.2 Admin Management

- Centralized dashboard for managing menus, orders, inventory, and user complaints.
- Add, update, remove menu items and adjust availability or discounts.
- Monitor and process orders; mark them as completed.
- Real-time inventory updates to optimize stock and reduce waste.
- Handle user complaints and update admin profile information.

ORDERBYTE ADMIN

MENUSORDERS

Menu Management

+ Add Item



















Search menu items...

All Categories

All Items

Menu Date: 20-04-2025

< PreviousTodayNext >

IMAGE	NAME	DESCRIPTION	CATEGORY	PRICE	STOCK	SPECIAL	ACTIONS
	7up	Lemon-lime flavored soft drink	Drinks	₹40.00	49 in stock	Today's Special	 
	Appam	Traditional South Indian pancake made from fermented rice batter	Breakfast	₹8.00	45 in stock	Today's Special	 
	Butterscotch Ice Cream	Sweet butterscotch flavored ice cream	Desserts	₹80.00	23 in stock	No	 
	Chapati	Whole wheat flatbread	Breakfast	₹10.00	Out of stock	No	 
	Chicken Biryani	Fragrant rice dish with spiced chicken and aromatics	Lunch	₹104.50 ₹110.00	25 in stock	No	 
							 

Last published: 4/20/2025, 2:05:16 PM

Publish Menu

Figure 4.5: OrderByte admin interface

4.6 User Interface

4.6.1 User Authentication

- Sign In: Login using email and password.
- Sign Up: Create a new account using email.
- Forgot Password: Reset password using email link.

4.6.2 Home Page

- Menu Categories: Large, clickable cards/tabs for food categories.
- Daily Exclusives: Featured daily specials displayed prominently at the top.
- View Details: Access full descriptions and pricing of menu items.
- Add to Cart: Pick the items you want to purchase and their quantity and place them in your shopping cart for later checkout.

4.6.3 Search and Filter

- Search food items using keywords.
- Filter based on category, availability, discounts, etc.
- Only currently available items are displayed to the user.

4.6.4 Cart and Checkout

- Review items added to the cart.
- Update quantities or delete items as needed.
- Continue to a secure payment process.
- Choose preferred payment method (e.g., credit/debit card, UPI, digital wallet).
- Proceed through a secure third-party payment gateway.
- Receive a detailed invoice and a unique order ID upon successful payment.

4.6.5 Profile Page

- Set or update your profile photo.
- Manage and update personal details such as phone number, name, etc.
- Access full history of past orders.

- Cancel orders that are still in progress or recently placed.

4.6.6 Complaint and Feedback

- Users can inform the admin about any problems or concerns they face while using the service or the platform.
- They can also submit suggestions to help improve the service.

4.6.7 Logout

- Allows users to securely sign out of their account.
- Ensures all session data is cleared to prevent unauthorized access.
- Helps maintain privacy, especially on shared or public devices.

4.7 Admin Interface

4.7.1 Admin Authentication

- Provides a secure login process for administrators using a unique Admin ID and password.
- Ensures that only authorized personnel can access the admin dashboard and manage system operations.

4.7.2 Menu Management

- Add new food options to the menu.
- Update item information, including availability, stock levels, and discounts.
- Remove outdated or unavailable items from the menu.
- Highlight selected dishes as part of the "Today's Menu" for customers to view.

4.7.3 Order Management

- Access a list of all customer orders placed on the current day.
- Track the status and progress of incoming orders in real time.

4.7.4 Complaint Management

- Access all complaints and feedback submitted by users.

- Respond to user concerns directly through the admin panel.
- Mark complaints as resolved or pending for better management

4.7.5 Profile Management

- Set or update your profile image.
- Edit personal details such as name and phone number.

4.8 System Flow

1. User selects items and places an order.
2. Demo payment is processed.
3. Admin manually updates order status.

This streamlined approach provides a functional and easy-to-use system for both users and admins, removing complexities like real-time tracking while maintaining efficient order management.

CHAPTER 5

DEVELOPMENT

Below are the details outlining the development process for OrderByte, our innovative college canteen management solution. The development phase is critical for converting our conceptual framework and requirements into a fully functional, user-friendly platform that improves the food ordering experience within educational institutions.

- a. Requirement Analysis: Identify and analyse the needs of students, faculty, and canteen staff to ensure the platform addresses all key operational challenges.
- b. Design: Create a detailed plan that outlines the system's architecture, user interface, and integration with real-time data services and other platforms.
- c. Development: Build the solution using robust backend frameworks and integrate a real-time database to ensure smooth data synchronization and efficient order management.
- d. Testing: Conduct comprehensive tests to confirm that the system is responsive, accurately tracks orders, and performs reliably under various conditions.
- e. Deployment: Launch the platform in a scalable and dependable manner, ensuring consistent accessibility for all users.
- f. Monitoring and Maintenance: Continuously monitor system performance, address any issues promptly, and update menus, inventory management, and integrations regularly to maintain peak performance.

5.1 Requirement Analysis

A comprehensive study was conducted to capture the needs of key users, including students, canteen personnel, and other stakeholders. Data gathered from surveys, focus groups, and interviews revealed major challenges such as long wait times, food wastage, and inefficient order processing. As a result, the system must incorporate essential features like real-time order tracking, dynamic menu updates, and effective inventory management. Users require seamless ordering experience with immediate confirmations, while administrators need an intuitive dashboard for easy menu management and stock monitoring. The goal is to design a solution that streamlines food ordering, reduces waste, and enhances service quality. Additionally, the platform must be scalable to handle increased demand and integrate future technological improvements.

5.2 Technology Selection

1. **Python IDE:** OrderByte is developed primarily in Python, utilizing its extensive libraries and frameworks for backend development. A dedicated Python IDE is essential for writing and testing code, with popular choices including PyCharm, Visual Studio Code, and Jupyter Notebook, all of which offer user-friendly interfaces and powerful development tools.
2. **Visual Studio Code (VS Code):** VS Code is a lightweight and flexible code editor that supports Python through various extensions. It provides features such as linting, debugging, and code snippets, enhancing development productivity. This editor can be easily downloaded from Microsoft's official website.
3. **Advanced Natural Language Processing:** OrderByte integrates cutting-edge natural language processing capabilities by leveraging both OpenAI's ChatGPT-4 and Anthropic's Claude AI. This dual integration enables the system to deliver intelligent, context-aware responses and interactive support, enriching the user experience with more intuitive and responsive conversational interfaces.
4. **Firebase:** OrderByte leverages Firebase to enable real-time data synchronization. As a cloud-hosted NoSQL database, Firebase offers secure, scalable storage that ensures instant updates for menu changes, order tracking, and inventory management.
5. **Python Flask:** The backend of OrderByte is built using Python Flask, a minimalistic web framework that streamlines web application development. Flask efficiently handles routing, HTTP requests, and API integrations, facilitating smooth communication between the user interface and the database.
6. **User Authentication and Password Management:** OrderByte includes secure user authentication features, including a forgot password and reset password functionality. This ensures that users can regain access to their accounts securely in case they forget their credentials. By implementing email or OTP-based verification, the system enhances security while providing a seamless recovery process.
7. **RazorPay (Third-Party Payment Integration):** OrderByte integrates Razorpay to facilitate secure and seamless online payments. This third-party payment gateway supports various payment methods, including credit/debit cards, UPI, and digital wallets. Transactions are processed in real time, ensuring a smooth checkout experience for users while maintaining security and reliability.

5.3 Design

Once the requirements are gathered, the design phase focuses on defining how the system will function and appear to the users. The design includes:

- **System Architecture:** A cloud-based architecture with Firebase for real-time data synchronization and Python Flask for backend development.
- **UI/UX Design:** The interface is designed to be intuitive, with a clear navigation structure and responsive layouts that work across various devices.
- **Database Design:** A structured database schema is created for storing user data, orders, menu items, inventory, and other critical information.
- **Security Design:** Measures for encryption, access control, and secure data storage are planned.

This phase results in wireframes, data flow diagrams, and a detailed project plan.

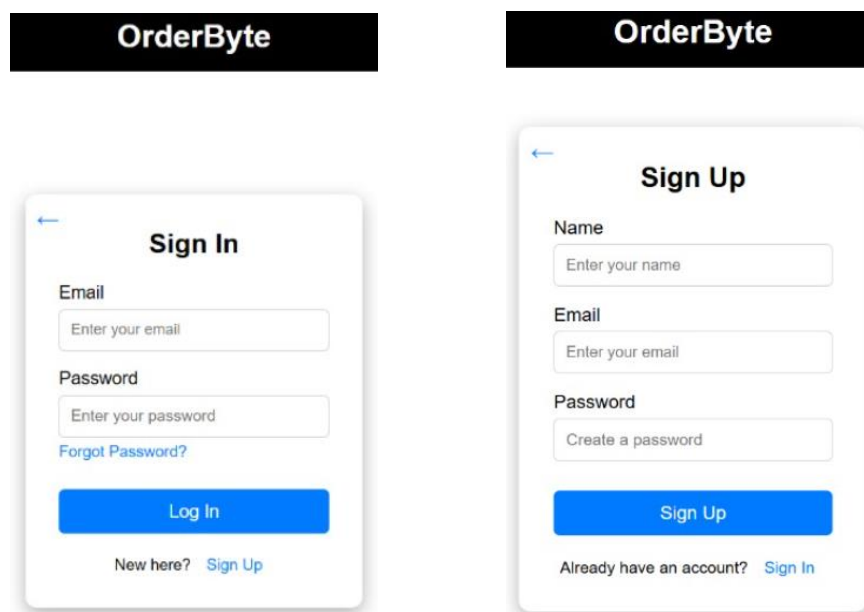


Figure 5.1: Sign-In and Sign-Up pages of OrderByte

5.4 Development

In the development phase, the actual coding of the OrderByte Smart College Canteen System begins. This stage includes:

- **Frontend Development:** Using HTML, CSS, and JavaScript to build an interactive and responsive web interface for users to browse the menu, place orders, and view order history.

- **Backend Development:** Implementing server-side logic using Python Flask to handle user authentication, order processing, and communication with the Firebase database.
- **Admin Dashboard Development:** Building an admin interface for menu management, order tracking, and inventory updates.

The development is done iteratively, with regular testing and adjustments to meet the requirements defined in earlier stages.

5.5 Testing

Once the system is developed, it undergoes rigorous testing to ensure it meets all functional, performance, and security requirements. This includes:

- **Functional Testing:** Verifying that users can browse the menu, place orders, and complete payments. Ensuring that admins can update menus and manage orders.
- **Performance Testing:** Evaluating the system's ability to handle multiple users simultaneously and under high traffic conditions.
- **Security Testing:** Ensuring that user data and payment information are securely processed and stored.

Any issues identified during this phase are addressed before moving to the next stage. The development is done iteratively, with regular testing and adjustments to meet the requirements defined in earlier stages.

5.6 Deployment

After successful testing, the system is deployed to a cloud-based server or local hosting environment. Deployment steps include:

- **Setting up the Hosting Environment:** Configuring the chosen cloud service (e.g., AWS, Google Cloud) or a dedicated server.
- **Database Deployment:** Connect and migrate the Firebase database to the live production environment for real-time data access.
- **Third-Party Payment Integration:** Securely integrate Razorpay or the selected payment service into the live system to enable smooth and reliable transactions.
- **Production Launch:** Making the system available to users, with proper monitoring in place to track performance and user activity.

5.7 Technology Stack

The OrderByte Smart College Canteen System uses the following technology stack:

- Frontend: HTML, CSS, JavaScript
- Backend: Python Flask
- Database: Firebase
- Payment Gateway: Stripe, Razorpay, or PayPal for secure transactions
- Hosting: AWS, Google Cloud, or dedicated server hosting
- Version Control: GitHub or GitLab for code management

5.8 System Implementation

The development of the OrderByte system follows a step-by-step approach to ensure a smooth and effective deployment. The first phase concentrates on building essential features, such as user registration, menu browsing, and order placement. These foundational features enable users to sign up, view available food options, and place orders with ease. After the core system is stable, more advanced functionalities are gradually added, including the integration of a secure payment gateway, the creation of an admin dashboard for restaurant management, and performance enhancements to improve the overall user experience. Each phase undergoes comprehensive testing to ensure reliability and user satisfaction before progressing to the next.

5.9 Error Handling Mechanism

Error handling is a crucial part of the system to ensure it operates smoothly even in the event of failures. The OrderByte Smart College Canteen System includes the following error handling mechanisms:

- User-Friendly Error Messages: Informing users of issues (e.g., payment failure, order processing delay) with clear instructions on how to resolve them.
- Backend Error Logging: Capturing server-side errors and logging them for analysis. These logs help identify potential issues like database failures or API issues.
- Fallback Mechanisms: In case of payment failure or other critical issues, the system provides an alternative flow (e.g., retry options or manual order management for admins).
- Automated Alerts: The system alerts administrators to critical errors or downtime, enabling them to take swift corrective actions.

5.10 Monitoring and Maintenance

Continuous monitoring and regular maintenance are essential to ensure the system operates smoothly. This includes

- **System Monitoring:** Keeping track of server health, response times, and user interactions to quickly address any issues that may arise.
- **Bug Fixes and Updates:** Regularly updating the system to fix bugs, improve features, and ensure compatibility with new technologies.
- **Performance Optimization:** Ongoing adjustments to improve system performance, such as optimizing database queries and scaling resources as needed.

This development process ensures that the OrderByte Smart College Canteen System is built in a methodical, user-centered, and efficient manner, addressing all the needs of the target users while ensuring scalability, security, and maintainability.

CHAPTER 6

TESTING & MAINTANENCE

The testing and maintenance phase is crucial to ensuring the OrderByte Smart College Canteen System operates smoothly, efficiently, and securely over time. This chapter outlines the various testing processes conducted to validate the functionality, performance, and security of the system, as well as the ongoing maintenance required to keep the system running optimally.

The system is designed to address common issues in traditional canteen management, such as long wait times, order inaccuracies, and food wastage. Therefore, rigorous testing was performed to confirm that it meets user needs and performs well under varying conditions. Maintenance strategies are also discussed to ensure the system continues to evolve and adapt to user requirements, address potential bugs, and implement performance enhancements.

By conducting comprehensive testing and establishing a robust maintenance plan, we aim to deliver a seamless, user-friendly experience for students, faculty, and visitors of the college canteen.

6.1 Testing Process

6.1.1 Functional Testing

Functional testing ensures that the OrderByte Smart College Canteen System meets functional requirements and performs expected tasks. The following steps were undertaken during functional testing:

- **User Interactions:** Verifying that users can browse the menu, add items to the cart, and complete payments.
- **Order Processing:** Ensuring that the system correctly updates order statuses (e.g., pending, completed, rejected) in real-time.
- **Role-Based Access Control:** Ensuring different roles (admin and users) have appropriate permissions for accessing features.
- **System Uptime:** Testing system availability and reliability to prevent downtime.

6.1.2 Performance Testing

Performance testing evaluates the system's responsiveness, stability, and ability to handle user load. The following steps were conducted:

- **Response Time Evaluation:** Measuring how quickly the system loads menus, processes orders, and updates statuses.
- **Concurrency Testing:** Ensuring the system can handle multiple simultaneous users ordering food without delays.
- **Resource Utilization Monitoring:** Checking CPU, memory, and database performance under varying user loads.
- **Stress Testing:** Identifying the system's breaking point under extreme traffic and assessing its recovery mechanism.

6.1.3 Security Testing

Security testing ensures that user data and transactions are protected. The security measures include:

- **Input Validation:** Preventing malicious inputs through secure form validation.
- **Encryption:** Ensuring payment data and user credentials are encrypted both in transit and at rest.
- **Access Control Testing:** Verifying that different user roles have restricted access based on their permissions.
- **Vulnerability Scanning:** Running automated security scans to detect and fix potential vulnerabilities.

6.2 Testing Results

6.2.1 Functional Issues

- **Cart Update Delays:** Some users experienced lag when modifying cart items. This was fixed by optimizing Firebase database calls.
- **Order Status Sync Issues:** Order statuses were occasionally not updated in real-time. The issue was resolved by improving backend synchronization using Python Flask and Firebase.

6.2.2 Performance Issues

- **Slow Load Times:** The system slowed under heavy traffic. Database queries were optimized, and caching mechanisms were introduced.

- **Queue Processing Delays:** Order queues were not updating promptly during peak hours. A queue optimization algorithm was implemented to resolve this.

6.2.3 Interface Issues

- **Navigation Difficulties:** Some users found the menu navigation unclear. UI adjustments were made to improve accessibility.
- **Menu Visibility Issues:** In some cases, the menu would not display correctly after page reloads. This was fixed by updating the front-end state management in JavaScript.

6.3 Maintenance

1. **Regular System Monitoring:** Continuous tracking of response times, error rates, and resource usage to prevent potential failures.
2. **Bug Fixes and Updates:** Ongoing debugging and feature enhancements based on user reports and performance logs.
3. **Performance Optimization:** Continuous performance tuning, including database optimizations, code refactoring, and load balancing improvements.
4. **Content Updates:** Regular menu updates, pricing adjustments, and new promotional offers added to ensure the system remains relevant and engaging.
5. **Security Enhancements:** Regular security patches and updates to protect user data and ensure compliance with security best practices.

This structured approach to testing and maintenance ensures that OrderByte Smart College Canteen System remains efficient, secure, and user-friendly.

CHAPTER 7

CONCLUSION

OrderByte is a comprehensive digital solution developed to modernize and streamline canteen operations in educational institutions. Traditional canteen systems often struggle with issues such as long queues, food wastage, delayed service, and inefficient order management. These challenges negatively affect the overall dining experience for students, faculty, and visitors. OrderByte addresses these concerns by introducing an intelligent, automated, and real-time system that facilitates efficient food ordering, better resource utilization, and enhanced communication between users and administrators.

The system is built using a powerful tech stack that includes Python Flask for backend operations and Firebase for real-time data synchronization. Frontend technologies like HTML, CSS, and JavaScript ensure that the user interface is responsive and visually appealing across various devices. This seamless integration of tools results in a fast, dynamic, and intuitive platform that delivers a satisfying experience to all stakeholders. Real-time updates ensure that menu availability, order status, and inventory changes are instantly reflected on both user and admin dashboards, eliminating miscommunication and service delays.

A standout feature of OrderByte is its user-focused design, which incorporates several enhancements to improve usability and convenience. Users can browse categorized menus organized into breakfast, lunch, drinks, desserts, and curries, helping them quickly find their preferred meals. Advanced search and filter options make the discovery process faster and more efficient. The cart system allows users to manage selected items, make changes, and proceed to checkout using integrated third-party payment gateways. This ensures secure, cashless, and fast transactions, making the overall process both modern and user-friendly.

The platform also supports meal pre-booking, allowing users to reserve meals ahead of time, an especially valuable feature for those with tight schedules or long commutes. Once the order is placed, users receive immediate confirmation and can track the status of their orders in real time. Additionally, the system enables users to view their order history and even cancel orders when necessary, adding a layer of control and flexibility. Profile management features allow users to update personal information such as phone numbers, email, and profile picture to personalize their experience. To further enhance service quality, users can submit complaints directly through the platform, ensuring quick communication with administrators and promoting accountability.

On the administrative side, OrderByte offers a centralized and real-time dashboard equipped with multiple management tools. Administrators can add, edit, or remove menu items, publish daily menus, and adjust food availability based on stock levels. Real-time inventory control helps prevent food wastage by aligning food preparation with actual demand. The admin panel also displays all active and completed orders, allowing staff to monitor and process them efficiently. In addition, administrators have access to a complaint management system where they can review and respond to user concerns, leading to continuous service improvements and greater user satisfaction.

Administrators also benefit from profile management features that allow them to maintain their own contact information and upload profile images for better identification within the system. These combined features streamline canteen operations, minimize manual workload, and eliminate common issues associated with traditional food service systems. As a result, canteen staff can focus more on delivering quality service rather than managing logistical challenges.

OrderByte is designed not only for functionality but also for scalability and sustainability. The system architecture supports expansion to multiple canteen locations, larger educational institutions, and even the addition of new modules such as dietary tracking, allergen filters, and AI-powered meal recommendations. Its reliance on cloud-based infrastructure reduces the need for extensive on-premises hardware, lowering operational costs and supporting eco-friendly practices. The real-time nature of the system allows for accurate demand forecasting, which directly contributes to the reduction of food waste and more responsible resource utilization.

Research and real-world studies emphasize the growing importance of smart canteen solutions that incorporate cloud computing, automation, and user-centered design. By digitizing the entire food ordering and management process, OrderByte fulfills this vision and sets a benchmark for future-ready dining services in educational environments. It addresses the key challenges of speed, efficiency, user satisfaction, and sustainability—all within a single, unified platform.

In conclusion, OrderByte offers an innovative and complete approach to enhancing canteen services in educational institutions. With its robust feature set, real-time capabilities, and intuitive interface, the platform not only solves the longstanding problems of traditional canteens but also prepares institutions for future scalability and technological advancement. By reducing wait times, automating manual tasks, minimizing food waste, and improving communication, OrderByte delivers a smart, scalable, and sustainable solution that benefits students, faculty, canteen staff, and institution administrators alike.

CHAPTER 8

REFERENCES

- [1] Lalitha, V., et al. "E-Canteen Management System based on Web Application." *2022 International Conference on Communication, Computing and Internet of Things (IC3IoT)*. IEEE, 2022.
- [2] Xu, Zixuan, et al. "The design of nutrition ordering system without queuing in the college canteen." *2020 International Conference on Computer Information and Big Data Applications (CIBDA)*. IEEE, 2020.
- [3] Ardiansyah, Muhammad Rizqi, et al. "The Website-Based Food Booking System Equipped with Real-Time Booking Status to Address Queuing Issues at Restaurants." *2021 International Conference on Electrical and Information Technology (IEIT)*. IEEE, 2021.
- [4] Avhad, Prashant, et al. "Canteen automation system with payment gateway." *Proceedings of the 3rd International Conference on Advances in Science & Technology (ICAST)*. 2020.
- [5] Gosalia, Jay, et al. "MERN-Based Canteen Management System: IoT Integration for RFID Payment and Table Reservation." *Available at SSRN 4802301* (2024).

CHAPTER 9

APPENDIX

PROGRAM

1. Initial imports, calls and configuration

```
from flask import Flask, jsonify, render_template, request, redirect,
url_for, flash, session, send_file
import firebase_admin
from firebase_admin import credentials, auth, db
import os
from dotenv import load_dotenv
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
import secrets
from datetime import datetime, timedelta
from functools import wraps
import requests
import re
import logging
import json
import uuid
from werkzeug.utils import secure_filename
from datetime import datetime
import io
import random
import string
import time
from werkzeug.utils import secure_filename # Make sure this is also imported

# Load environment variables
load_dotenv()
# Initialize Flask app
app = Flask(__name__)
app.secret_key = os.getenv("SECRET_KEY", "CS-B")
app.config['SERVER_NAME'] = 'localhost:5000'
ADMIN_EMAIL = os.getenv("ADMIN_EMAIL", "admin@gmail.com")
FIREBASE_API_KEY = os.getenv("FIREBASE_API_KEY") # Add this to your .env file
app.config['UPLOAD_FOLDER'] = 'static/uploads'
os.makedirs(app.config['UPLOAD_FOLDER'], exist_ok=True)

# Initialize Firebase Admin SDK
cred = credentials.Certificate("serviceAccountKey.json")
firebase_admin.initialize_app(cred, {
```



```

        'databaseURL': 'https://orderbyte-2025-default-rtdb.firebaseio.com' # Your
Realtime Database URL
    })
    ref = db.reference('/')

# Add this to app.py after Firebase initialization
def get_today_date():
    return datetime.now().strftime('%Y-%m-%d')

```

2. Initialization of items in the menu

```

# Initialize menu items if they don't exist
# Update the initialize_menu_items function in app.py
def initialize_menu_items():
    menu_items_ref = db.reference('menu_items')
    existing_items = menu_items_ref.get()
    if not existing_items:
        # Define default menu items
        default_items = {
            # Breakfast items (existing)
            "appam": {
                "name": "Appam",
                "meal_type": "breakfast",
                "price": 8,
                "image_url": "/static/img/appam.png"
            },
            "masala-dosa": {
                "name": "Masala Dosa",
                "meal_type": "breakfast",
                "price": 80,
                "image_url": "/static/img/dosa.webp"
            },
            "idli-sambar": {
                "name": "Idli Sambar (4 pcs)",
                "meal_type": "breakfast",
                "price": 30,
                "image_url": "/static/img/idli.webp"
            },
            "poori-masala": {
                "name": "Poori Masala",
                "meal_type": "breakfast",
                "price": 30,
                "image_url": "/static/img/poori.jpg"
            },
            "chapati": {
                "name": "Chapati",
                "meal_type": "breakfast",

```

```

        "price": 10,
        "image_url": "/static/img/chapati.jpg"
    },
    "puttu": {
        "name": "Puttu",
        "meal_type": "breakfast",
        "price": 15,
        "image_url": "/static/img/puttu.jpg"
    },
    "upma": {
        "name": "Upma",
        "meal_type": "breakfast",
        "price": 20,
        "image_url": "/static/img/upma.jpg"
    },
    # Lunch items (existing)
    "veg-thali": {
        "name": "Veg Thali",
        "meal_type": "lunch",
        "price": 50,
        "image_url": "/static/img/vegmeals.webp"
    },
    "chicken-biryani": {
        "name": "Chicken Biryani",
        "meal_type": "lunch",
        "price": 110,
        "image_url": "/static/img/chickenbiryani.jpg"
    },
    "fish-curry-meals": {
        "name": "Fish Curry Meals",
        "meal_type": "lunch",
        "price": 70,
        "image_url": "/static/img/fishthali.jpg"
    },
    "veg-pulao": {
        "name": "Veg Pulao",
        "meal_type": "lunch",
        "price": 60,
        "image_url": "/static/img/vegpulao.jpg"
    },
    "poratta": {
        "name": "Poratta",
        "meal_type": "lunch",
        "price": 10,
        "image_url": "/static/img/poratta.jpeg"
    },
    "chapati": {
        "name": "Chapati",

```

```

        "meal_type": "lunch",
        "price": 10,
        "image_url": "/static/img/chapati.jpg"
    },
    # Add drinks & desserts items (new)
    "masala-chai": {
        "name": "Masala Chai",
        "meal_type": "drinks",
        "price": 30,
        "image_url": "/static/img/masalachai.jpg"
    },
    "gulab-jamun": {
        "name": "Gulab Jamun (2 pcs)",
        "meal_type": "desserts",
        "price": 50,
        "image_url": "/static/img/gulabjamun.webp"
    },
    "mango-lassi": {
        "name": "Mango Lassi",
        "meal_type": "drinks",
        "price": 40,
        "image_url": "/static/img/mangolassi.webp"
    },
    "vanilla-icecream": {
        "name": "Vanilla IceCream",
        "meal_type": "desserts",
        "price": 80,
        "image_url": "/static/img/vanilla.jpg"
    },
    "chocolate-icecream": {
        "name": "Chocolate IceCream",
        "meal_type": "desserts",
        "price": 80,
        "image_url": "/static/img/chocolate.webp"
    },
    "butterscotch-icecream": {
        "name": "Butterscotch IceCream",
        "meal_type": "desserts",
        "price": 80,
        "image_url": "/static/img/butterscotch.jpg"
    },
    "strawberry-icecream": {
        "name": "Strawberry IceCream",
        "meal_type": "desserts",
        "price": 80,
        "image_url": "/static/img/strawberry.jpg"
    },
    "pepsi": {

```

```

        "name": "Pepsi",
        "meal_type": "drinks",
        "price": 40,
        "image_url": "/static/img/pepsi.jpg"
    },
    "7up": {
        "name": "7up",
        "meal_type": "drinks",
        "price": 40,
        "image_url": "/static/img/7up.jpg"
    },
    "coca-cola": {
        "name": "Coca-Cola",
        "meal_type": "drinks",
        "price": 40,
        "image_url": "/static/img/cocacola.webp"
    },
    "frooti": {
        "name": "Frooti",
        "meal_type": "drinks",
        "price": 40,
        "image_url": "/static/img/frooti.webp"
    },
    # Add curry items (new)
    "veg-curry": {
        "name": "Veg Curry",
        "meal_type": "curry",
        "price": 50,
        "image_url": "/static/img/vegcurry.jpg"
    },
    "chicken-curry": {
        "name": "Chicken Curry",
        "meal_type": "curry",
        "price": 120,
        "image_url": "/static/img/chickencurry.jpg"
    },
    "fish-curry": {
        "name": "Fish Curry",
        "meal_type": "curry",
        "price": 110,
        "image_url": "/static/img/fishcurry.jpg"
    },
    "kadala-curry": {
        "name": "Kadala Curry",
        "meal_type": "curry",
        "price": 40,
        "image_url": "/static/img/kadalacurry.jpg"
    },
    },

```

```

        "egg-curry": {
            "name": "Egg Curry",
            "meal_type": "curry",
            "price": 40,
            "image_url": "/static/img/eggcurry.jpg"
        }
    }
    # Save default items to database
    menu_items_ref.set(default_items)
    print("Initialized default menu items")

```

3. Creating Admin User

```

# Create admin user during initialization if it doesn't exist
def ensure_admin_exists():
    try:
        # Check if admin user exists
        admin_user = auth.get_user_by_email(ADMIN_EMAIL)
        print(f"Admin user exists: {admin_user.uid}")
        # Ensure admin role is set in database
        admin_ref = db.reference('admins').child(admin_user.uid)
        if not admin_ref.get():
            admin_ref.set({'email': ADMIN_EMAIL, 'role': 'admin'})
            print("Admin role updated in database")

    except auth.UserNotFoundError:
        # Create admin user if not exists
        admin_password = os.getenv("ADMIN_PASSWORD", "strong-admin-password")
        admin_user = auth.create_user(
            email=ADMIN_EMAIL,
            password=admin_password,
            display_name="Administrator"
        )
        # Save admin data in Realtime Database
        admin_data = {
            'name': "Administrator",
            'email': ADMIN_EMAIL,
            'uid': admin_user.uid
        }
        # Save in users collection
        user_ref = db.reference('users')
        user_ref.child(admin_user.uid).set(admin_data)
        # Save in admins collection
        admin_ref = db.reference('admins')
        admin_ref.child(admin_user.uid).set({
            'email': ADMIN_EMAIL,
            'role': 'admin'
        })

```

```

        print(f"Admin user created: {admin_user.uid}")
    except Exception as e:
        print(f"Error ensuring admin exists: {str(e)}")

# Call function to ensure admin exists during startup
ensure_admin_exists()
# Admin required decorator
def admin_required(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        if 'user_id' not in session:
            flash('Please sign in to access this page.', 'error')
            return redirect(url_for('signin'))

        # Check if user is an admin
        admin_ref = db.reference(f'admins/{session["user_id"]}')
        admin_data = admin_ref.get()
        if not admin_data or admin_data.get('role') != 'admin':
            flash('You do not have permission to access this page.', 'error')
            return redirect(url_for('home'))
        return f(*args, **kwargs)
    return decorated_function

```

4. Index Route

```

# Home Route (Index Page)
@app.route('/')
def index():
    return render_template('index.html')

```

5. Sign-In and Sign-Up Route

```

# Sign Up Route
@app.route('/signup', methods=['GET', 'POST'])
def signup():
    if request.method == 'POST':
        email = request.form['email']
        password = request.form['password']
        name = request.form['name']

        try:
            # Create user in Firebase Authentication
            user = auth.create_user(
                email=email,
                password=password,
                display_name=name
            )

```

```

        print(f"User created in Firebase Authentication: {user.uid}")

        # Save user data in Realtime Database
        user_data = {
            'name': name,
            'email': email,
            'uid': user.uid
        }
        ref = db.reference('users')
        ref.child(user.uid).set(user_data)
        print(f"User data saved in Realtime Database: {user_data}")

        flash('Account created successfully! Please sign in.', 'success')
        return redirect(url_for('signin'))
    except Exception as e:
        print(f"Error: {str(e)}")
        flash(f'Error: {str(e)}', 'error')
        return redirect(url_for('signup'))
    return render_template('signup.html')

# Sign In Route - FIXED with proper authentication
@app.route('/signin', methods=['GET', 'POST'])
def signin():
    if request.method == 'POST':
        email = request.form['email']
        password = request.form['password']

        try:
            # Use Firebase Auth REST API to verify credentials
            auth_url =
"https://identitytoolkit.googleapis.com/v1/accounts:signInWithPassword"
            if not FIREBASE_API_KEY:
                flash('Server configuration error: Missing API key', 'error')
                return redirect(url_for('signin'))

            payload = {
                "email": email,
                "password": password,
                "returnSecureToken": True
            }

            response = requests.post(f"{auth_url}?key={FIREBASE_API_KEY}",
json=payload)
            data = response.json()

            if 'error' in data:
                error_message = data['error'].get('message', 'Authentication
failed')

```

```

        # Translate Firebase error messages to user-friendly messages
        if error_message == 'EMAIL_NOT_FOUND':
            flash('No account found with that email address.', 'error')
        elif error_message == 'INVALID_PASSWORD':
            flash('Incorrect password. Please try again.', 'error')
        elif error_message == 'USER_DISABLED':
            flash('This account has been disabled.', 'error')
        else:
            flash(f'Sign in error: {error_message}', 'error')

        return redirect(url_for('signin'))

    # If we get here, authentication was successful
    # Get the user from Firebase Admin SDK for additional information
    user = auth.get_user_by_email(email)
    # Store user ID in session
    session['user_id'] = user.uid

    # Check if user is an admin
    admin_ref = db.reference(f'admins/{user.uid}')
    admin_data = admin_ref.get()
    if admin_data and admin_data.get('role') == 'admin':
        session['is_admin'] = True
        flash('Admin sign in successful!', 'success')
        return redirect(url_for('menuadmin'))
    else:
        session['is_admin'] = False
        flash('Sign in successful!', 'success')
        return redirect(url_for('home'))

except requests.RequestException as e:
    print(f"Network error: {str(e)}")
    flash('Network error. Please check your connection and try again.',
'error')
    return redirect(url_for('signin'))
except Exception as e:
    print(f"Error: {str(e)}")
    flash(f'Error during sign in: {str(e)}', 'error')
    return redirect(url_for('signin'))
return render_template('signin.html')

```

6. Home or Dashboard Route

```

# Home/Dashboard Route
@app.route('/home')
def home():
    # Check if user is logged in

```



```

if 'user_id' not in session:
    flash('Please sign in to access this page.', 'error')
    return redirect(url_for('signin'))

# Fetch user data from Realtime Database
user_id = session['user_id']
ref = db.reference(f'users/{user_id}')
user_data = ref.get()
if not user_data:
    flash('User data not found.', 'error')
    return redirect(url_for('signin'))

return render_template('home.html', user=user_data)

@app.route('/admin')
@admin_required
def menuman():
    # Fetch all users from the database
    users_ref = db.reference('users')
    users = users_ref.get()
    return render_template('menuman.html', users=users)

```

7. Logout Route

```

# Logout Route
@app.route('/logout')
def logout():
    # Clear the session
    session.pop('user_id', None)
    session.pop('is_admin', None)
    session.pop('_flashes', None)
    flash('You have been logged out.', 'success')
    session.clear()
    return redirect(url_for('index'))

def single_flash(message, category):
    # Clear existing flashes
    session.pop('_flashes', None)
    # Set the new flash message
    flash(message, category)

```

8. Forgot Password Route

```

# Forgot Password Route
@app.route('/forgot-password', methods=['GET', 'POST'])
def forgot_password():

```

```

if request.method == 'POST':
    email = request.form['email']

    try:
        # Check if the email exists in Firebase Authentication
        try:
            user = auth.get_user_by_email(email)
            uid = user.uid # Get the user ID
        except auth.UserNotFoundError:
            flash('No account found with that email address.', 'error')
            return redirect(url_for('forgot_password'))

        # Generate a token
        token = secrets.token_urlsafe(32)
        # Store the reset token in the Realtime Database
        ref = db.reference('password_reset_tokens')
        expiry_time = datetime.now() + timedelta(hours=1)

        reset_data = {
            'token': token,
            'email': email,
            'uid': uid,
            'expiry': expiry_time.timestamp(),
            'used': False
        }

        # Use sanitized email as the key
        safe_email = email.replace('.', '_')
        ref.child(safe_email).set(reset_data)
        # Generate reset link using url_for instead of base_url
        # We need to use _external=True to get a full URL
        reset_link = url_for('reset_password', uid=uid, token=token,
_external=True)
        print(f"Generated Reset Link: {reset_link}")
        # Send email with reset link
        email_sent = send_reset_email(email, reset_link)

        if email_sent:
            flash('Password reset email sent! Check your inbox.', 'success')
        else:
            flash('Error sending email. Please try again later.', 'error')

        return redirect(url_for('signin'))
    except Exception as e:
        print(f"Error in forgot_password: {str(e)}")
        flash(f'Error: {str(e)}', 'error')
        return redirect(url_for('forgot_password'))
return render_template('forgot-password.html')

```

9. Reset Password Route

```
# Reset Password Route
@app.route('/reset-password', methods=['GET', 'POST'])
def reset_password():
    uid = request.args.get('uid')
    token = request.args.get('token')
    if not uid or not token:
        flash('Invalid password reset link.', 'error')
        return redirect(url_for('signin'))

    try:
        # Get user email from uid
        user = auth.get_user(uid)
        email = user.email
        safe_email = email.replace('.', '_')
        # Verify token from database using safe_email
        ref = db.reference(f'password_reset_tokens/{safe_email}')
        reset_data = ref.get()

        if not reset_data or reset_data.get('token') != token or
reset_data.get('used'):
            flash('Invalid or expired reset token.', 'error')
            return redirect(url_for('signin'))

        # Check if token is expired
        expiry_time = datetime.fromtimestamp(reset_data.get('expiry', 0))
        if datetime.now() > expiry_time:
            flash('Reset link has expired. Please request a new one.', 'error')
            return redirect(url_for('forgot_password'))

        if request.method == 'POST':
            new_password = request.form['new_password']
            confirm_password = request.form['confirm_password']

            if new_password != confirm_password:
                flash('Passwords do not match!', 'error')
                return render_template('reset-password.html', uid=uid,
token=token, email=email)
            try:
                # Update the password in Firebase Authentication
                auth.update_user(
                    uid,
                    password=new_password
                )
                # Mark token as used
                ref.update({'used': True})
```

```

        flash('Password updated successfully! Please sign in with your
new password.', 'success')
        return redirect(url_for('signin'))
    except Exception as e:
        flash(f'Error updating password: {str(e)}', 'error')
        return render_template('reset-password.html', uid=uid,
token=token, email=email)

    return render_template('reset-password.html', uid=uid, token=token,
email=email)

except Exception as e:
    print(f"Error in reset_password: {str(e)}")
    flash('Error validating reset request.', 'error')
    return redirect(url_for('signin'))

```

10. Menu Management

```

@app.route('/get-today-menu', methods=['GET'])
def get_today_menu():
    try:
        # Get date from query parameter
        date = request.args.get('date')
        print(f"Requested menu for date: {date}") # Debug log

        if not date:
            # If no date provided, use today's date
            today = datetime.now()
            date = today.strftime('%Y%m%d')
            print(f"No date provided, using today: {date}") # Debug log

        # Use db.reference instead of db.child
        menu_ref = db.reference(f'menus/{date}')
        menu_data = menu_ref.get()
        print(f"Retrieved data: {menu_data}") # Debug log

        if not menu_data:
            return jsonify({
                'message': 'No menu found for the requested date',
                'menu': None
            })
        # Return the menu data
        return jsonify({
            'message': 'Menu retrieved successfully',
            'menu': menu_data
        })

```

```

except Exception as e:
    print(f"Error in get_today_menu: {str(e)}") # Debug log
    return jsonify({
        'error': str(e),
        'message': 'Failed to retrieve menu'
    }), 500
@app.route('/menu-management', methods=['GET', 'POST'])
@admin_required
def menu_management():
    if request.method == 'POST':
        try:
            # Get the data from the request
            data = request.get_json()
            date = data.get('date')
            selected_items = data.get('selectedItems', [])
            status = data.get('status', 'draft') # Default to draft if not
specified

            if not date:
                return jsonify({'error': 'Date is required'}), 400

            # Format date for database key (remove hyphens)
            date_key = date.replace('-', '')
            # Create reference to menus in database
            menu_ref = db.reference(f'menus/{date_key}')
            # Group items by meal type
            breakfast_items = []
            lunch_items = []
            drinks_desserts_items = []
            curry_items = []
            for item in selected_items:
                item_id = item.get('id')

                # Create a new dictionary with proper fields for each item
                item_data = {
                    'id': item_id,
                    'name': item.get('name'),
                    'price': item.get('price'),
                    'image_url': item.get('image_url'),
                    'available': True
                }

                # Determine meal type based on item ID or other criteria
                if item_id in ['appam', 'masala-dosa', 'idli-sambar', 'poori-
masala', 'puttu', 'upma'] or (item_id == 'chapati' and 'breakfast' in
item.get('meal_type', '')):
                    breakfast_items.append(item_data)

```

```

        elif item_id in ['veg-thali', 'chicken-biryani', 'fish-curry-meals', 'veg-pulao', 'poratta'] or (item_id == 'chapati' and 'lunch' in item.get('meal_type', '')):
            lunch_items.append(item_data)
        elif item_id in ['masala-chai', 'gulab-jamun', 'mango-lassi', 'vanilla-icecream', 'chocolate-icecream', 'butterscotch-icecream', 'strawberry-icecream', 'pepsi', '7up', 'coca-cola', 'frooti']:
            drinks_desserts_items.append(item_data)
        elif item_id in ['veg-curry', 'chicken-curry', 'fish-curry', 'kadala-curry', 'egg-curry']:
            curry_items.append(item_data)
        else:
            # Check meal_type attribute if available
            meal_type = item.get('meal_type', '')
            if 'breakfast' in meal_type:
                breakfast_items.append(item_data)
            elif 'lunch' in meal_type:
                lunch_items.append(item_data)
            elif 'drinks_desserts' in meal_type:
                drinks_desserts_items.append(item_data)
            elif 'curry' in meal_type:
                curry_items.append(item_data)
            else:
                # Default to lunch if not specified
                lunch_items.append(item_data)
    # Save to database with all sections
    menu_data = {
        'date': date,
        'status': status,
        'breakfast': breakfast_items,
        'lunch': lunch_items,
        'drinks_desserts': drinks_desserts_items,
        'curry': curry_items,
        'last_updated': datetime.now().isoformat()
    }
    menu_ref.set(menu_data)
    return jsonify({'success': True, 'message': 'Menu saved successfully'})

except Exception as e:
    print(f"Error saving menu: {str(e)}")
    return jsonify({'error': str(e)}), 500

# GET request - return current menu for the specified date
date = request.args.get('date')
if not date:
    return jsonify({'error': 'Date is required'}), 400
date_key = date.replace('-', '')

```

```

menu_ref = db.reference(f'menus/{date_key}')
existing_menu = menu_ref.get()
# Also fetch all available menu items
all_items_ref = db.reference('menu_items')
all_items = all_items_ref.get() or {}

return jsonify({
    'existing_menu': existing_menu,
    'all_items': all_items
})

@app.route('/menu-items', methods=['GET'])
def get_all_menu_items():
    try:
        # Get all menu items from the database
        items_ref = db.reference('menu_items')
        all_items = items_ref.get()
        if not all_items:
            return jsonify({
                'message': 'No menu items found',
                'items': {}
            })

        return jsonify({
            'message': 'Menu items retrieved successfully',
            'items': all_items
        })

    except Exception as e:
        print(f"Error retrieving menu items: {str(e)}")
        return jsonify({
            'error': str(e),
            'message': 'Failed to retrieve menu items'
        }), 500

```

11. Cart Management

```

# Cart Route
@app.route('/add-to-cart', methods=['POST'])
def add_to_cart():
    # Check if user is logged in
    if 'user_id' not in session:
        return jsonify({
            'success': False,
            'message': 'Please sign in to add items to your cart'
        }), 401
    try:

```

```

# Get data from request
data = request.get_json()
item_id = data.get('item_id')
name = data.get('name')
price = data.get('price')
image_url = data.get('image_url')
quantity = data.get('quantity', 1)

if not all([item_id, name, price, image_url]):
    return jsonify({
        'success': False,
        'message': 'Invalid item data'
    }), 400
user_id = session['user_id']
today_date = get_today_date()
# Reference to user's cart in Firebase for today's date
cart_ref = db.reference(f'carts/{user_id}/dates/{today_date}/items')
# Check if item already exists in cart
existing_item = cart_ref.child(item_id).get()
if existing_item:
    # Update quantity if item already exists
    new_quantity = existing_item.get('quantity', 0) + quantity
    cart_ref.child(item_id).update({
        'quantity': new_quantity,
        'last_updated': datetime.now().isoformat()
    })
else:
    # Add new item to cart
    cart_ref.child(item_id).set({
        'id': item_id,
        'name': name,
        'price': price,
        'image_url': image_url,
        'quantity': quantity,
        'date_added': datetime.now().isoformat(),
        'last_updated': datetime.now().isoformat()
    })

# Get updated cart total
cart_items = cart_ref.get() or {}
item_count = sum(item.get('quantity', 0) for item in cart_items.values())
cart_total = sum(item.get('price', 0) * item.get('quantity', 0) for item
in cart_items.values())

# Update current cart reference
db.reference(f'carts/{user_id}/current_date').set(today_date)
return jsonify({
    'success': True,

```



```

        'message': 'Item added to cart',
        'cart': {
            'item_count': item_count,
            'total': cart_total,
            'items': cart_items
        }
    })

except Exception as e:
    # Log the error
    app.logger.error(f"Error adding item to cart: {str(e)}")
    return jsonify({
        'success': False,
        'message': 'An error occurred while adding item to cart'
    }), 500

# Add these routes to your app.py file
@app.route('/cart')
def cart():
    """Render the cart page"""
    return render_template('cart.html')

@app.route('/get-cart-items', methods=['GET'])
def get_cart_items():
    """Get all items in the user's cart for today"""
    # Check if user is logged in
    if 'user_id' not in session:
        return jsonify({
            'success': False,
            'message': 'Please sign in to view your cart'
        }), 401

    try:
        user_id = session['user_id']
        today_date = get_today_date()

        # Update current cart date
        db.reference(f'carts/{user_id}/current_date').set(today_date)
        # Reference to user's cart in Firebase for today's date
        cart_ref = db.reference(f'carts/{user_id}/dates/{today_date}/items')
        # Get all items in the cart
        cart_items = cart_ref.get() or {}
        # Calculate cart total
        cart_total = sum(item.get('price', 0) * item.get('quantity', 0) for item
in cart_items.values())

        return jsonify({
            'success': True,
            'cart': {

```

```

        'items': cart_items,
        'total': cart_total,
        'item_count': sum(item.get('quantity', 0) for item in
cart_items.values()),
        'date': today_date
    }
})

except Exception as e:
    app.logger.error(f"Error retrieving cart items: {str(e)}")
    return jsonify({
        'success': False,
        'message': 'An error occurred while retrieving cart items'
    }), 500

@app.route('/update-cart-item', methods=['POST'])
def update_cart_item():
    """Update the quantity of an item in the cart"""
    # Check if user is logged in
    if 'user_id' not in session:
        return jsonify({
            'success': False,
            'message': 'Please sign in to update your cart'
        }), 401

    try:
        data = request.get_json()
        item_id = data.get('item_id')
        quantity_change = data.get('quantity_change', 0)
        if not item_id or quantity_change == 0:
            return jsonify({
                'success': False,
                'message': 'Invalid request data'
            }), 400
        user_id = session['user_id']
        today_date = get_today_date()
        # Reference to the specific item in user's cart
        item_ref =
db.reference(f'carts/{user_id}/dates/{today_date}/items/{item_id}')

        # Get current item data
        item_data = item_ref.get()
        if not item_data:
            return jsonify({
                'success': False,
                'message': 'Item not found in cart'
            }), 404
        # Calculate new quantity

```

```

        new_quantity = item_data.get('quantity', 0) + quantity_change

        # If quantity becomes zero or negative, remove the item
        if new_quantity <= 0:
            item_ref.delete()
        else:
            # Update the quantity
            item_ref.update({
                'quantity': new_quantity,
                'last_updated': datetime.now().isoformat()
            })
            # Update item data for response
            item_data['quantity'] = new_quantity
        # Get updated cart for total calculation
        cart_ref = db.reference(f'carts/{user_id}/dates/{today_date}/items')
        cart_items = cart_ref.get() or {}

        # Calculate new cart total
        cart_total = sum(item.get('price', 0) * item.get('quantity', 0) for item
in cart_items.values())
        return jsonify({
            'success': True,
            'message': 'Cart updated successfully',
            'item': item_data,
            'cart': {
                'items': cart_items,
                'total': cart_total,
                'item_count': sum(item.get('quantity', 0) for item in
cart_items.values())
            }
        })
    except Exception as e:
        app.logger.error(f"Error updating cart item: {str(e)}")
        return jsonify({
            'success': False,
            'message': 'An error occurred while updating cart'
        }), 500

@app.route('/remove-cart-item', methods=['POST'])
def remove_cart_item():
    """Remove an item from the cart"""
    # Check if user is logged in
    if 'user_id' not in session:
        return jsonify({
            'success': False,
            'message': 'Please sign in to modify your cart'
        }), 401
    try:

```

```

data = request.get_json()
item_id = data.get('item_id')

if not item_id:
    return jsonify({
        'success': False,
        'message': 'Invalid request data'
    }), 400

user_id = session['user_id']
today_date = get_today_date()

# Reference to the specific item in user's cart
item_ref =
db.reference(f'carts/{user_id}/dates/{today_date}/items/{item_id}')

# Check if item exists
if not item_ref.get():
    return jsonify({
        'success': False,
        'message': 'Item not found in cart'
    }), 404

# Remove the item
item_ref.delete()

# Get updated cart for total calculation
cart_ref = db.reference(f'carts/{user_id}/dates/{today_date}/items')
cart_items = cart_ref.get() or {}

# Calculate new cart total
cart_total = sum(item.get('price', 0) * item.get('quantity', 0) for item
in cart_items.values())

return jsonify({
    'success': True,
    'message': 'Item removed from cart',
    'cart': {
        'items': cart_items,
        'total': cart_total,
        'item_count': sum(item.get('quantity', 0) for item in
cart_items.values())
    }
})
except Exception as e:
    app.logger.error(f"Error removing cart item: {str(e)}")
    return jsonify({
        'success': False,
        'message': 'An error occurred while removing item from cart'
    }), 500

```

12. Searching Code

```
@app.route('/search', methods=['GET'])
def search():
    """Render the search page and handle search functionality"""
    try:
        # Get date from query parameter or use today's date
        date = request.args.get('date')
        if not date:
            today = datetime.now()
            date = today.strftime('%Y%m%d')
        else:
            # Format date for database key (remove hyphens if present)
            date = date.replace('-', '')
        # Fetch today's menu
        menu_ref = db.reference(f'menus/{date}')
        menu_data = menu_ref.get()
        # Fetch all menu items
        all_items_ref = db.reference('menu_items')
        all_items = all_items_ref.get() or {}
        # Convert all_items from dictionary to list and add an 'in_stock' field
        all_items_list = []
        # Track which items are in today's menu
        in_menu_items = set()

        if menu_data:
            # Collect all item IDs that are in today's menu
            for category in ['breakfast', 'lunch', 'drinks_desserts', 'curry']:
                if category in menu_data and menu_data[category]:
                    for item in menu_data[category]:
                        if 'id' in item:
                            in_menu_items.add(item['id'])

        # Process all items and mark their availability
        for item_id, item_data in all_items.items():
            item = dict(item_data) # Create a copy
            item['id'] = item_id
            item['in_stock'] = item_id in in_menu_items
            all_items_list.append(item)
        # Organize items by meal type
        organized_items = {
            'breakfast': [],
            'lunch': [],
            'drinks': [],
            'desserts': [],
            'curry': []
        }
        for item in all_items_list:
```

```

        meal_type = item.get('meal_type', '')
        if meal_type in organized_items:
            organized_items[meal_type].append(item)
        elif meal_type == 'drinks_desserts':
            # Split between drinks and desserts based on name
            if any(term in item.get('name', '').lower() for term in ['chai',
'lassi', 'pepsi', 'cola', '7up', 'frooti']):
                organized_items['drinks'].append(item)
            else:
                organized_items['desserts'].append(item)

    # Prepare menu data for template
    formatted_menu = None
    menu_date = None
    if menu_data:
        formatted_menu = {
            'breakfast': menu_data.get('breakfast', []),
            'lunch': menu_data.get('lunch', []),
            'drinks_desserts': menu_data.get('drinks_desserts', []),
            'curry': menu_data.get('curry', [])
        }
        menu_date = menu_data.get('date', date)
    # Handle conversion of formatted date if needed
    if menu_date and len(menu_date) == 8: # Format YYYYMMDD
        year = menu_date[:4]
        month = menu_date[4:6]
        day = menu_date[6:8]
        menu_date = f"{year}-{month}-{day}"

    return render_template('search.html',
                           menu=formatted_menu,
                           menu_date=menu_date,
                           all_items=organized_items)
except Exception as e:
    print(f"Error in search: {str(e)}")
    flash(f'Error retrieving menu data: {str(e)}', 'error')
    return render_template('search.html', menu=None, all_items={})

```

13. Checkout

```

@app.route('/checkout')
def checkout():
    # Check if user is logged in
    if 'user_id' not in session:
        flash('Please sign in to checkout.', 'error')
        return redirect(url_for('signin'))

```

```

user_id = session['user_id']
today_date = get_today_date()

# Fetch cart data
cart_ref = db.reference(f'carts/{user_id}/dates/{today_date}/items')
cart_items = cart_ref.get() or {}
# Calculate total
cart_total = sum(item.get('price', 0) * item.get('quantity', 0) for item in
cart_items.values())

if not cart_items:
    flash('Your cart is empty.', 'info')
    return redirect(url_for('search'))
return render_template('checkout.html', cart_items=cart_items,
cart_total=cart_total)

@app.route('/order-confirmation')
def order_confirmation():
    # Check if user is logged in
    if 'user_id' not in session:
        flash('Please sign in to view order confirmation.', 'error')
        return redirect(url_for('signin'))
    order_id = request.args.get('order_id')
    if not order_id:
        flash('No order specified.', 'error')
        return redirect(url_for('search'))

    # Fetch order details
    order_ref = db.reference(f'orders/{order_id}')
    order_data = order_ref.get()
    # Check if order exists and belongs to the current user
    if not order_data or order_data.get('user_id') != session['user_id']:
        flash('Order not found or unauthorized.', 'error')
        return redirect(url_for('search'))
    # Get order date from order_data, with fallbacks if not present
    order_date = order_data.get('order_date')
    if not order_date:
        # Try using created_at if available, otherwise use current time
        if 'created_at' in order_data:
            # If created_at is an ISO format string, convert it to readable
format
            from datetime import datetime
            try:
                created_datetime =
datetime.fromisoformat(order_data['created_at'])
                order_date = created_datetime.strftime("%B %d, %Y %I:%M %p")
            except (ValueError, TypeError):
                # If conversion fails, use the raw string

```

```

        order_date = order_data['created_at']
    else:
        # Last resort: use current time
        from datetime import datetime
        order_date = datetime.now().strftime("%B %d, %Y %I:%M %p")

    # Calculate order details for the template
    total_items = sum(item.get('quantity', 0) for item in order_data.get('items',
{})).values()
    total_amount = order_data.get('total', 0)

    # Render the template with all necessary order information
    return render_template('order_confirmation.html',
                           order_id=order_data['order_id'],
                           order_date=order_date,
                           items=order_data.get('items', {}),
                           total_items=total_items,
                           total_amount=total_amount,
                           status=order_data.get('status', 'processing'))

@app.route('/process-payment', methods=['POST'])
def process_payment():
    # Check if user is logged in
    if 'user_id' not in session:
        return jsonify({
            'success': False,
            'message': 'Please sign in to complete your order'
        }), 401
    try:
        user_id = session['user_id']
        today_date = get_today_date()

        # xGet cart data
        cart_ref = db.reference(f'carts/{user_id}/dates/{today_date}/items')
        cart_items = cart_ref.get() or {}

        if not cart_items:
            return jsonify({
                'success': False,
                'message': 'Cart is empty'
            }), 400

        # Process payment data
        data = request.get_json()
        # Generate order ID
        order_id = str(uuid.uuid4())
        # Save order to database
        order_data = {
            'order_id': order_id,

```



```

        'user_id': user_id,
        'items': cart_items,
        'total': sum(item.get('price', 0) * item.get('quantity', 0) for item
in cart_items.values()),
        'payment_method': data.get('payment_method', 'card'),
        'status': 'completed',
        'created_at': datetime.now().isoformat(),
        'order_date': datetime.now().strftime("%B %d, %Y %I:%M %p") # Add
this line
    }

    # Save to orders collection
    orders_ref = db.reference('orders')
    orders_ref.child(order_id).set(order_data)
    # Also save to user's orders
    user_orders_ref = db.reference(f'users/{user_id}/orders')
    user_orders_ref.child(order_id).set(order_data)
    # Clear the cart after successful order
    cart_ref.delete()

    return jsonify({
        'success': True,
        'order_id': order_id,
        'message': 'Payment processed successfully'
    })

except Exception as e:
    app.logger.error(f"Error processing payment: {str(e)}")
    return jsonify({
        'success': False,
        'message': 'An error occurred while processing payment'
    }), 500

def is_admin(user_id):
    """Check if a user is an admin"""
    if not user_id:
        return False

    try:
        db = firebase_admin.db.reference()
        user_ref = db.child('users').child(user_id) # Consistent with other
functions
        user_data = user_ref.get()
        return user_data and user_data.get('is_admin', False)
    except Exception as e:
        app.logger.error(f"Admin check error: {str(e)}")
        return False

```

14. User Management

```
# Add this context processor to make admin_data available to all templates
@app.context_processor
def inject_admin_data():
    return {'admin_data': get_admin_data()}
# Update the profile route to handle admin profile specifically
@app.route('/api/admin/profile', methods=['POST'])
@admin_required
def update_admin_profile():
    try:
        admin_id = session.get('user_id')
        if not admin_id:
            return jsonify({'error': 'Not authenticated'}), 401

        # Get form data
        name = request.form.get('name')
        email = request.form.get('email')
        phone = request.form.get('phone')
        # Update admin data in the admins table
        admin_ref = db.reference('admins').child(admin_id)
        updates = {
            'name': name,
            'email': email,
            'phone': phone
        }

        # Handle profile picture upload if present
        if 'profile_picture' in request.files:
            file = request.files['profile_picture']
            if file and file.filename:
                # Create a secure filename
                filename = secure_filename(file.filename)
                # Create unique filename with timestamp
                unique_filename = f"{admin_id}_{int(time.time())}_{filename}"
                file_path = os.path.join(app.config['UPLOAD_FOLDER'],
unique_filename)
                # Save the file
                file.save(file_path)
                # Update the profile picture path
                updates['profile_picture'] = f"/static/uploads/{unique_filename}"

        # Update the admin data in the database
        admin_ref.update(updates)
        # Also update the corresponding user record if it exists
        user_ref = db.reference('users').child(admin_id)
        user_data = user_ref.get()
        if user_data:
            user_ref.update(updates)
```

```

        return jsonify({'success': True})
    except Exception as e:
        app.logger.error(f"Error updating admin profile: {str(e)}")
        return jsonify({'error': str(e)}), 500

@app.route('/api/user/<user_id>')
def get_user(user_id):
    try:
        # Check if user_id is "admin" (special case)
        if user_id == "admin":
            # Try to get the actual admin user ID from the session
            actual_user_id = session.get('user_id')
            if actual_user_id:
                user_id = actual_user_id
            else:
                # Get the admin user from admins collection
                admins_ref = db.reference('admins')
                admins = admins_ref.get()
                if admins:
                    # Get the first admin ID
                    admin_id = list(admins.keys())[0]
                    user_id = admin_id

        # Get user data
        user_ref = db.reference('users').child(user_id)
        user_data = user_ref.get()

        if user_data:
            # Add the user_id to the response so client knows the actual ID
            user_data['id'] = user_id
            return jsonify(user_data)
        return jsonify({'error': 'User not found'}), 404
    except Exception as e:
        app.logger.error(f"Error fetching user data: {str(e)}")
        return jsonify({'error': 'Server error'}), 500

@app.route('/profile')
def profile():
    if 'user_id' not in session:
        return redirect(url_for('login'))
    user_id = session['user_id']
    user_data = ref.child(f'users/{user_id}').get()

    # Get user's orders
    orders_ref = ref.child('orders')
    orders = orders_ref.get()
    user_orders = []
    if orders:

```

```

        for order_id, order_data in orders.items():
            if order_data.get('user_id') == user_id:
                # Get the timestamp
                timestamp = order_data.get('created_at') or
order_data.get('order_date', '')

                # Format the date in a user-friendly way
                formatted_date = "N/A"
                if timestamp:
                    try:
                        # Convert ISO timestamp to datetime object
                        from datetime import datetime
                        dt = datetime.fromisoformat(timestamp.replace("Z",
"+00:00"))

                        # Format it as DD/MM/YYYY
                        formatted_date = dt.strftime("%d/%m/%Y")
                    except:
                        # If there's any error in parsing, use the original
                        formatted_date = timestamp
                # Format the order data properly for the template
                formatted_order = {
                    'order_id': order_id,
                    'date': formatted_date,
                    'amount': order_data.get('total', '0.00')
                }
                user_orders.append(formatted_order)

    return render_template('profile.html', user_id=user_id, user_data=user_data,
orders=user_orders)

# Add this function to get user data for use in all templates
def get_user_data():
    user_id = session.get('user_id') # Changed from 'user' to 'user_id'
    user_data = None
    if user_id:
        user_ref = db.reference('users').child(user_id) # Fixed Firebase
reference
        user_data = user_ref.get() or {}
        # Add default profile picture if none exists
        if 'profile_picture' not in user_data or not
user_data['profile_picture']:
            user_data['profile_picture'] = '/static/uploads/default_user.png'
    return user_data

# Add this context processor to make user_data available to all templates
@app.context_processor
def inject_user_data():
    return {'user_data': get_user_data()}

```

```

@app.route('/api/user/<user_id>/profile', methods=['POST'])
def update_user_profile(user_id):
    try:
        # Check if user_id is "admin" (special case)
        if user_id == "admin":
            # Try to get the actual admin user ID from the session
            actual_user_id = session.get('user_id')
            if actual_user_id:
                user_id = actual_user_id
            else:
                # Get the admin user from admins collection
                admins_ref = db.reference('admins')
                admins = admins_ref.get()
                if admins:
                    # Get the first admin ID
                    admin_id = list(admins.keys())[0]
                    user_id = admin_id

        # First get the existing user data to preserve all fields
        user_ref = db.reference('users').child(user_id)
        existing_user_data = user_ref.get() or {}

        # Create updates object, preserving existing data
        updates = existing_user_data.copy()
        # Get form data and update only if provided
        if 'name' in request.form and request.form['name']:
            updates['name'] = request.form['name']
        if 'email' in request.form and request.form['email']:
            updates['email'] = request.form['email']
        if 'phone' in request.form and request.form['phone']:
            updates['phone'] = request.form['phone']

        # Handle profile picture upload if present
        if 'profile_picture' in request.files:
            file = request.files['profile_picture']
            if file and file.filename:
                # Create a secure filename
                filename = secure_filename(file.filename)
                # Create unique filename with timestamp
                unique_filename = f"{user_id}_{int(time.time())}_{filename}"
                file_path = os.path.join(app.config['UPLOAD_FOLDER'],
unique_filename)
                # Save the file
                file.save(file_path)
                # Update the profile picture path
                updates['profile_picture'] = f"/static/uploads/{unique_filename}"

        # Update the user data in the database

```

```

        user_ref.set(updates) # Use set instead of update to ensure complete
data update

    # Return updated profile data for UI refresh
    return jsonify({
        'success': True,
        'data': {
            'profile_picture': updates.get('profile_picture',
'/static/uploads/default_user.png')
        }
    })
except Exception as e:
    app.logger.error(f"Error updating user profile: {str(e)}")
    return jsonify({'success': False, 'error': str(e)}), 500
@app.route('/api/user/<user_id>/orders', methods=['GET'])
def get_user_orders(user_id):
    if 'user_id' not in session or session['user_id'] != user_id:
        return jsonify({'success': False, 'error': 'Unauthorized'}), 401

    # Get user's orders
    orders_ref = ref.child('orders')
    orders = orders_ref.get()
    user_orders = []
    if orders:
        for order_id, order_data in orders.items():
            if order_data.get('user_id') == user_id:
                # Get the timestamp
                timestamp = order_data.get('created_at') or
order_data.get('order_date', '')

                # For sorting purposes
                sort_value = 0
                # Format the date in a user-friendly way
                formatted_date = "N/A"
                if timestamp:
                    try:
                        # Convert ISO timestamp to datetime object
                        from datetime import datetime
                        dt = datetime.fromisoformat(timestamp.replace("Z",
"+00:00"))

                        # Use timestamp as epoch time for reliable sorting
                        sort_value = dt.timestamp()
                        # Format it as DD/MM/YYYY
                        formatted_date = dt.strftime("%d/%m/%Y")
                    except:
                        # If there's any error in parsing, use the original
                        formatted_date = timestamp
                # Get items data from the order

```

```

items = []
if 'items' in order_data:
    if isinstance(order_data['items'], dict):
        # If items is a dictionary, extract names or product_ids
        for item_id, item_data in order_data['items'].items():
            if isinstance(item_data, dict):
                item_name = item_data.get('name', item_id)
                items.append(item_name)
            else:
                items.append(str(item_data))
    elif isinstance(order_data['items'], list):
        # If items is already a list
        items = order_data['items']
    else:
        # If items is a string or other type
        items = [str(order_data['items'])]

# Format the order data properly for the frontend
formatted_order = {
    'order_id': order_id,
    'date': formatted_date,
    'amount': order_data.get('total', '0.00'),
    'items': items,
    '_sort_value': sort_value # For sorting, not sent to client
}
user_orders.append(formatted_order)

# Sort orders by timestamp value (most recent first)
user_orders.sort(key=lambda x: x.get('_sort_value', 0), reverse=True)
# Remove sort value before sending to client
for order in user_orders:
    if '_sort_value' in order:
        del order['_sort_value']

return jsonify({'success': True, 'orders': user_orders})

```