**Payment Gateway**

**(Payment Gateway Integration System)**

**A PROJECT REPORT**

**for**

**Mini-Project 2 (ID201B)**

**Session (2024-25)**

**Submitted by**

**Nilesh Agrahari**

202410116100134

**Nishant Raikwar**

202410116100135

**Preet Kumar**

202410116100147

**Rabita Yadav**

202410116100156

**Submitted in partial fulfilment of the**

**Requirements for the Degree of**

**MASTER OF COMPUTER APPLICATION**

**Under the Supervision of**

**Dr. Vipin Kumar**

### Associate Professor



**Submitted to**

**Department Of Computer Applications**

**KIET Group of Institutions, Ghaziabad**

**Uttar Pradesh-201206**

**(APRIL- 2025)**

**CERTIFICATE**

Certified that **Nilesh Agrahari 202410116100134, Nishant Raikwar 202410116100135, Preet Kumar 202410116100147 and Rabita Yadav 202410116100135** has/have carried out the project work having “**Payment Gateway Integration System**” MINI PROJECT - 2 (FULL STACK DEVELOPMENT) (ID201B)) for **Master of Computer Application** from Dr. A.P.J. Abdul Kalam Technical University (AKTU) (formerly UPTU), Lucknow under my supervision. The project report embodies original work, and studies are carried out by the student himself/herself and the contents of the project report do not form the basis for the award of any other degree to the candidate or to anybody else from this or any other University/Institution.

This is to certify that the above statement made by the candidate is correct to the best of my knowledge.

Date:

**Dr. Vipin Kumar Dr. Akash Rajak**

**Associate Professor Dean**

**Department of Computer Applications Department of Computer Applications**

**KIET Group of Institutions, Ghaziabad KIET Group of Institutions, Ghaziabad**

**An Autonomous Institution An Autonomous Institution**

**Payment Gateway**

**ABSTRACT**

In the current digital era, where online transactions have become the cornerstone of e-commerce and digital services, integrating secure and seamless payment systems is not just a luxury but a necessity. This project, titled **"Payment Gateway Integration"**, is developed to showcase the end-to-end implementation of an online payment system using **Razorpay**, a widely used and developer-friendly payment gateway **API**, within a **Django** web application framework.

The project demonstrates the secure handling of customer payments, starting from order creation to payment processing, signature verification, and post-payment order management.

Using Django’s structured MVC approach, the backend efficiently manages data such as order creation, Razorpay order ID handling, and secure transaction logging. The frontend uses Razorpay’s checkout widget to provide a smooth user experience with support for multiple payment methods, including UPI, credit/debit cards, and net banking.

The application not only serves as a real-time prototype for payment integration in business systems (like e-commerce, online services, or canteen management), but it also offers insights into best practices in payment verification, secure data flow, and integration of third-party APIs with web applications. This project bridges the gap between theoretical learning and practical implementation of secure financial transaction processing systems.

**Keywords:**  
Payment Gateway Integration, Razorpay API, Django Payment Integration, Secure Online Transactions, Razorpay Django Project, Payment Gateway Project in Django.

**ACKNOWLEDGEMENT**

Success in life is never attained single-handedly. My deepest gratitude goes to my project supervisor, **Dr. Vipin Kumar** for his guidance, help, and encouragement throughout my project work. Their enlightening ideas, comments, and suggestions.

Words are not enough to express my gratitude to Dr. Akash Rajak, Professor and Dean, Department of Computer Applications, for his insightful comments and administrative help on various occasions.

Fortunately, I have many understanding friends, who have helped me a lot on many critical conditions.

Finally, my sincere thanks go to my family members and all those who have directly and indirectly provided me with moral support and other kind of help. Without their support, completion of this work would not have been possible in time. They keep my life filled with enjoyment and happiness.

**Nilesh Agrahari**

**Nishant Raikwar**

**Preet Kumar**

**Rabita Yadav**

**TABLE OF CONTENT**

Certificate ii

Abstract iii

Acknowledgements iv

Table of Contents v

List of Abbreviations 5

List of Figures 6

1. **Introduction 7-29**
   1. Problem in Current System 7
   2. Proposed Solution 8
   3. Functional Requirements 10
   4. Non-Functional Requirements 11
2. **Feasibility Study**  **30-36**
3. **Project Objective** **37**
4. **Hardware and Software Requirements** **40**
5. **Project Flow 51-62**
   1. Project Flow 18
   2. Modules 19
   3. ER Diagram 22
   4. Data Flow Diagram 26
6. **Project Outcome** **63-68**
7. **References** **69**
8. **Bibliography 70**

**LIST OF ABBREVIATION**

|  |  |  |
| --- | --- | --- |
| **S.No.** | **Full Name of Abbreviation** | **Abbreviation** |
| 1 | Payment Gateway Integration | PGI |
| 2 | Unified Payments Interface | UPI |
| 3 | Application Programming Interface | API |
| 4 | Hyper Text Transfer Protocol Secure | HTTPS |
| 5 | Secure Sockets Layer | SSL |
| 6 | Representational State Transfer | REST |
| 7 | JavaScript Object Notation | JSON |
| 8 | Database Management System | DBMS |
| 9 | Structured Query Language | SQL |
| 10 | User Interface | UI |
| 11 | User Experience | UX |
| 12 | Customer Relationship Management | CRM |
| 13 | Payment Card Industry Data Security Standard | PCI DSS |
| 14 | Know Your Customer | KYC |
| 15 | Transaction Reference Number | TRN |
| 16 | Software Development Kit | SDK |
| 17 | Content Management System | CMS |
| 18 | Web Application Framework | WAF |
| 19 | Two-Factor Authentication | 2FA |
| 20 | Virtual Private Network | VPN |

**LIST OF FIGURES**

|  |  |  |  |
| --- | --- | --- | --- |
| **Figure No.** | **Name of Figure** | **Page No.** | |
| 5.1 | Project Development Stage | | 51 | |
| 5.2 | System Module Overview | | 53 | |
| 5.3 | ER diagram | | 56 | |
| 5.4 | Zero level DFD | | 59 | |
| 5.5 | User Side DFD | | 60 | |
| 5.6 | Payment Integration | | 61 | |

### ****Chapter 1****

**INTRODUCTION**

The rise of digital platforms has revolutionized the way individuals and organizations carry out daily activities, including shopping, ordering food, booking services, and paying bills. One of the foundational elements of these digital experiences is the ability to facilitate **secure, real-time online payments**. With the ever-growing shift towards cashless transactions and e-commerce, integrating payment gateways into web applications has become a vital component of modern system architecture.

The **"Payment Gateway Integration"** project aims to implement a secure and functional payment solution within a Django-based web application. This project specifically focuses on incorporating the **Razorpay** payment gateway, leveraging its robust API for seamless order creation, user-friendly frontend checkout, and secure server-side verification using cryptographic signatures.

By simulating a real-world payment flow—starting from selecting an item, placing an order, initiating a payment, and verifying the success of the transaction—this project provides a hands-on demonstration of how secure digital payments can be integrated into custom-built web systems. It fills the gap between theoretical knowledge and practical application, preparing developers for real-world development scenarios in the e-commerce and service-based domains.

**Purpose of the System**

The primary purpose of this **Payment Gateway Integration system** is to enable secure, seamless, and scalable online transactions within a Django web application. In an increasingly digital economy, customers demand convenience, speed, and trust when paying for services or products online. By integrating Razorpay’s payment gateway into a web platform, this system offers an end-to-end solution for real-time payment processing, which is essential for modern e-commerce or service-based applications.

This system aims to eliminate the limitations of manual or offline transactions by automating order handling, streamlining payment authentication, and ensuring financial accuracy. Moreover, by managing the complete transaction lifecycle—from order initiation to server-side signature verification—it sets a robust foundation for integrating financial features into web platforms of all sizes.

**Target Audience**

While the system is currently designed as a demo application for educational and developmental use, it is structured in a modular and scalable way, making it applicable for real-world deployment in:

* E-commerce platforms
* Online food ordering systems
* Event ticket booking portals
* Subscription-based services
* Educational institutions (for fees or merchandise)
* Startups and tech businesses wanting to automate their payments

Its user interface and backend logic are built to be **developer-friendly**, while the frontend experience is intuitive enough for everyday users who may not be technically inclined. This dual approach ensures that the system is suitable both as a teaching tool and as a deployable product.

**Key Features**

**1.** **Secure Razorpay Checkout Integration**  
A fully-functional Razorpay checkout interface enables customers to complete payments using UPI, cards, wallets, or net banking—within a secure and responsive UI.

**2.** **Razorpay Orders API (Server-Side)**  
Orders are generated on the server using Razorpay’s authenticated API calls. Each transaction is assigned a unique razorpay\_order\_id ensuring traceability.

**3. Signature Verification**  
To prevent tampering, Razorpay returns a secure signature that is verified on the backend before marking any transaction as “Paid”.

**4.** **Database Logging and Order Tracking**  
The system saves complete transaction metadata including payment\_id, order\_id, signature, and payment status in the backend database.

**5.** **Error and Failure Handling**  
If the signature verification fails, the system marks the order as “Failed” and alerts the user, avoiding false positives and financial discrepancies.

**6.** **Extensible Admin Interface**  
Django’s admin panel allows authorized users to view order logs, filter transactions by status, and audit all payment activity in one place.

**Benefits**

**• For Customers:**

* Seamless online payment experience
* Trust and transparency in transactions
* Faster processing with instant feedback on payment success/failure

**• For Developers:**

* A practical implementation of third-party API integration
* Clear separation of frontend and backend logic for maintainability
* Ready-to-use payment infrastructure that can be scaled or reused

**• For Businesses:**

* Enhanced automation and operational efficiency
* Data collection on transaction history for analytics and insights
* Better compliance with digital payment standards

**Technological Foundation**

The project is developed using a proven, secure, and scalable technology stack:

| **Layer** | **Technology Used** | **Purpose** |
| --- | --- | --- |
| Backend | **Django (Python)** | MVC architecture, database management |
| Frontend | **HTML, CSS, JavaScript** | User interface and Razorpay checkout |
| Payment API | **Razorpay** | Secure payment processing and verification |
| Database | **SQLite (default in Django)** | Persistent storage for order and payment data |
| API Security | **HMAC SHA256** | Server-side payment response verification |

This stack is optimized for rapid development, modular expansion, and deployment readiness in small to mid-sized projects.

**Relevance in the Modern Context**

As businesses and institutions move toward **contactless, cashless, and digital transactions**, payment gateway integration is no longer optional—it is a competitive necessity. Users expect not only a service but also a **frictionless and trustworthy payment experience**. Whether it's buying products, paying service fees, or subscribing to a plan, the financial flow must be streamlined.

This project provides the foundational logic to meet these expectations while introducing best practices such as API usage, cryptographic verification, and real-time status management. Furthermore, by using Razorpay—a platform trusted by leading Indian startups and businesses—it aligns with real-world industry practices.

By building such a system, this project **bridges the gap between academic learning and industry demands**, helping developers gain exposure to technologies they are likely to encounter in production environments. It also lays the groundwork for future features like invoices, refunds, and subscriptions, making it a smart investment in long-term application architecture.

**1.1 Problem in Existing System**

Despite the increasing digitalization of services and commerce, many existing web-based platforms still **lack robust and secure payment solutions**. Whether it’s an online shop, a service portal, or an internal institutional tool, the absence of an integrated payment gateway leads to inefficiencies, poor user experience, and operational challenges. Below are the major issues identified in the traditional systems that do not utilize modern payment integration like Razorpay:

**1. Time-Consuming Manual Transactions**

Without a digital payment system in place, transactions are often handled offline—through cash, manual bank transfers, or post-order collection. These processes are not only slow and cumbersome but also introduce delays in order confirmation and service fulfillment. This affects both the user and the service provider, especially when dealing with high volumes of transactions.

**2. Error-Prone Order and Payment Tracking**

In systems lacking automation, keeping a consistent record of which orders have been paid and which are pending is a major challenge. This often results in order-processing errors, disputes over payments, and mismanagement of customer expectations. It becomes increasingly difficult to audit or verify payments accurately when they are tracked manually or across multiple disconnected tools.

**3. Limited User Convenience**

Modern users expect to complete their entire transaction online—from browsing to payment. A system that does not offer integrated digital payment forces users to rely on external apps, banking portals, or even physical visits, reducing convenience and leading to drop-offs or abandoned orders.

**4. Lack of Real-Time Confirmation**

Without real-time payment confirmation, users are often left wondering whether their payment was successful. There is no instant feedback loop to assure them that their transaction has gone through and that their service or order will be processed. This uncertainty results in repeated user queries, which increase the load on customer support teams.

**5. Dependency on Physical or Offline Payments**

Legacy systems usually depend on cash at delivery or card swipes at physical counters. This not only leads to long queues and delays but also increases the burden of handling physical currency, reconciliation errors, and security risks. In today’s digital world, this reliance on offline payment methods limits scalability and user satisfaction.

**6. Security Vulnerabilities**

When payment handling is done without a proper payment gateway, it often lacks encryption, secure communication, and fraud protection. Exchanging sensitive details like card numbers via insecure methods poses serious data privacy and security risks. Compliance with payment industry standards (like PCI-DSS) is nearly impossible without gateway integration.

**7. Lack of Transaction Analytics and Insights**

Without a proper payment backend, the system has no data on transaction volumes, peak usage periods, or user payment preferences. This lack of financial analytics hinders strategic planning, revenue tracking, and customer experience improvements. Businesses lose out on valuable insights that could help them scale or adapt.

**8. Not Suitable for Socially-Distanced, Contactless Environments**

In the post-pandemic world, contactless transactions have become essential. Systems without integrated digital payments require users to physically interact—whether it’s swiping cards, handing over cash, or visiting counters. This exposes both customers and staff to hygiene and health risks. Online systems with integrated payment gateways reduce the need for any physical interaction, making them more suitable for modern, health-conscious environments.

**1.2 Proposed System**

To overcome the challenges faced by traditional systems that rely on manual or loosely-coupled payment processes, this project proposes a **robust, secure, and real-time digital payment solution** using the **Razorpay Payment Gateway**, integrated directly into a Django-based web application.

This solution introduces a **complete payment lifecycle** that begins with order creation and ends with secure payment verification and backend transaction logging. Every step is carefully designed to ensure **data integrity, security, and a smooth user experience**.

**1. Secure Online Payment Integration with Razorpay**

The core of this system is the integration of **Razorpay**, a trusted and feature-rich payment gateway that supports multiple payment methods including UPI, credit and debit cards, net banking, and wallets.

By using Razorpay’s **Orders API**, the Django server generates a unique razorpay\_order\_id for every payment initiated. This order ID acts as a secure reference for the transaction, ensuring that each payment is traceable, verifiable, and uniquely tied to a specific order.

The Razorpay **Checkout.js widget** is embedded on the frontend, providing a highly interactive and secure interface for customers to complete their payments. This avoids the need for redirecting users to third-party sites, maintaining continuity and trust within the application.

**2. Server-Side Signature Verification**

Upon successful payment, Razorpay returns three crucial data points to the application:

* razorpay\_payment\_id
* razorpay\_order\_id
* razorpay\_signature

To ensure that the response hasn’t been tampered with during transmission, the server performs an **HMAC SHA256 verification** using the razorpay\_order\_id, razorpay\_payment\_id, and the Razorpay **secret key**. This cryptographic operation confirms that the response is genuinely from Razorpay and has not been altered in transit.

Only after this verification is passed does the system update the order status to **“Paid”**. If verification fails, the order is immediately flagged as **“Failed”**, and no further action is taken—ensuring maximum transaction integrity.

**3. Automated Order and Payment Status Handling**

The Django backend is responsible for:

* Creating and storing order records
* Saving payment IDs and signatures
* Verifying payment status
* Updating the payment flag (is\_paid)
* Notifying users of success/failure

This approach removes the burden of manual payment tracking and ensures that **every order is handled in a consistent and secure manner**.

**4. Enhanced User Experience and Real-Time Confirmation**

The frontend leverages JavaScript and Razorpay’s SDK to offer a **streamlined checkout experience**. Users are notified of successful transactions instantly, and confirmation messages are shown without reloading the page.

There is **no waiting period or manual confirmation** required. Customers receive real-time feedback and can trust that their payment has been successfully recorded.

This immediate confirmation enhances user satisfaction and encourages repeat usage.

**5. Scalable Architecture and Future Expansion**

The current implementation is modular and extensible. This means that:

* Additional features like **cart integration**, **discount codes**, or **subscriptions** can be easily added.
* Admin users can monitor all transactions through Django’s built-in admin panel.
* Payment data can be exported or analyzed using third-party tools for business insights.
* Security can be further enhanced by integrating two-factor authentication (2FA) or webhook-based verification for enterprise-grade use.

By following best practices in API integration, data management, and UI design, this system becomes a **future-ready solution** for any digital platform looking to accept payments securely and efficiently.

**Benefits of the Proposed System**

The integration of Razorpay into a Django application provides value across multiple levels: users, developers, and business administrators. Below are the categorized benefits based on different stakeholders:

**for End Users / Customers:**

* **Fast and Convenient Payments:** Users can complete transactions quickly using their preferred payment methods.
* **Improved Trust:** Secure checkout and real-time feedback increase user confidence.
* **Reduced Confusion:** Instant success/failure messages reduce uncertainty about payment status.

**For Developers / Admins:**

* **Simplified Implementation:** Razorpay’s SDK and Django’s architecture make integration straightforward.
* **Real-Time Payment Monitoring:** All transactions are logged and verifiable.
* **Built-in Error Handling:** Failed or unverified payments are automatically flagged, reducing errors.

**For Businesses / Project Stakeholders:**

* **Increased Efficiency:** Automates the entire payment process, reducing the need for manual reconciliation.
* **Better Decision-Making:** Transaction logs and status tracking enable analysis and reporting.
* **Scalable Solution:** Built with reusability and future integration (e.g., subscriptions, invoices) in mind.

**1.3 Functional Requirements**

Functional requirements define the specific behaviors, actions, and operations that the system must perform to meet the core objectives of the application. In the context of the **Payment Gateway Integration** project, these requirements focus on how users interact with the system to initiate payments, how the backend handles the logic of order and payment processing, and how Razorpay’s response is verified to ensure the authenticity of each transaction. These functionalities ensure that the system not only provides a convenient and secure experience for end users but also maintains integrity and traceability in financial transactions. By mapping out each functional component in detail, the development process becomes more structured, easier to test, and scalable for future enhancements. These requirements lay the foundation for a robust and production-ready payment infrastructure that can be adapted to various web applications requiring secure payment processing. Whether applied to a small web portal or a large e-commerce system, these functionalities are critical for ensuring the smooth flow of digital transactions, enhancing user trust, and minimizing errors throughout the process.

Below is a detailed list of the functional requirements for this system:

**FR1: User Can Initiate a Purchase**

The system must provide a clear, intuitive interface for users to select an item or service they wish to purchase. This interaction begins the entire payment process. Users should be able to view the product description, pricing, and any related details before confirming the intent to buy. Upon clicking the “Pay Now” or “Proceed to Payment” button, the system must capture the relevant purchase information (such as item ID, quantity, and total price) and route it to the backend to begin order generation.

It is important that this initiation process is smooth and responsive, with no unnecessary page reloads or delays. The system should also validate user inputs at this stage to ensure that the selected item is available, and the order data is complete. Any failure or missing information should be gracefully handled by displaying user-friendly messages prompting corrective action.

This functionality should support both desktop and mobile platforms, ensuring cross-device accessibility and consistent behavior regardless of the environment. The user initiation process forms the critical entry point to the payment gateway flow, and thus must be implemented with careful attention to usability and clarity.

**FR2: Server Creates Razorpay Order**

After receiving the purchase initiation from the frontend, the backend system—built in Django—must interact with Razorpay’s Order Creation API using the credentials provided in the settings file. The server must construct a properly formatted JSON payload containing the amount (in the smallest sub-unit, such as paise), currency (typically INR), and a unique receipt identifier for the transaction. This payload is then sent to Razorpay over HTTPS using the authenticated Razorpay Python SDK.

On successful creation, Razorpay responds with a structured JSON object containing a unique razorpay\_order\_id, along with other metadata like creation time, entity type, and status. This order ID must be captured and stored in the database as part of the Order model, along with all user and transaction-related data. It is essential to save this ID because it acts as a secure link between the client-side payment and the server-side verification process.

The system must also handle errors gracefully—if the Razorpay API fails or returns an error, the backend should notify the frontend and log the incident for future debugging. All communication with the Razorpay server should be encrypted and time-limited to avoid hanging requests or duplicate order creation.

**FR3: Razorpay Checkout Interface Is Rendered**

Once the server has generated a valid razorpay\_order\_id, this identifier—along with other dynamic details such as amount, currency, customer name, email, and contact number—must be passed to the Razorpay Checkout widget rendered on the frontend. This is achieved through the secure JavaScript SDK provided by Razorpay (checkout.js), which generates a modal-based interface for processing the payment.

The Razorpay widget should be customized to reflect the branding of the system, including the name of the business or application, logo, and theme color. It must offer multiple payment options (UPI, cards, wallets, net banking) in a way that is clear and easy to use. The interface should automatically close or refresh on payment success or failure, and allow users to cancel gracefully if needed.

Additionally, the checkout form should include optional fields for pre-filling customer information, improving convenience and reducing drop-offs. The system must ensure that no sensitive data, such as API secrets, are ever exposed in the frontend code. All sensitive configuration should be securely handled on the server side.

**FR4: Collect Razorpay Payment Response**

After the user completes a transaction through the Razorpay Checkout interface, Razorpay sends a payment response object to the frontend. This response contains three essential fields: razorpay\_order\_id, razorpay\_payment\_id, and razorpay\_signature. These values must be collected accurately and securely using JavaScript and then transmitted to the backend for verification.

The frontend must handle this submission via a POST request, using appropriate headers (such as CSRF tokens for Django security compliance) and ensure that no data tampering or interception can occur. The system should include robust error handling to capture cases where Razorpay does not return a response (e.g., payment cancelled or timeout) and inform the user accordingly.

This response object is the foundation for verifying the authenticity of the transaction, and hence it must be transmitted securely, without any modifications. Developers must ensure that the correct field names are used, and the payload is structured exactly as required by the backend verification process.

**FR5: Verify Razorpay Signature Server-Side**

The backend system must validate the authenticity of the payment using Razorpay’s server-side verification technique. This involves generating a signature using the HMAC SHA256 algorithm by concatenating the razorpay\_order\_id and razorpay\_payment\_id, and hashing it with the Razorpay KEY\_SECRET. The result is then compared with the razorpay\_signature received from the frontend.

If the two signatures match, the system can confirm that the payment is valid and has not been tampered with during transmission. If the signatures do not match, it indicates a potential fraud or network interception, and the transaction must be rejected. This validation step is **non-negotiable** and must be performed before updating the payment status in the database.

The server must log all verification attempts and their results, including timestamp, status, and any mismatches, to support audit trails and debugging. The logic should also handle edge cases, such as missing fields or malformed responses, to ensure robustness.

**FR6: Mark Order as Paid or Failed**

Following successful signature verification, the system must update the relevant order in the database by setting its is\_paid flag to True, and changing its status to “Success” or “Completed”. In the case of a failed or unverifiable signature, the order should be marked as “Failed”, and no further processing (such as order fulfillment or delivery) should take place.

This update must be atomic and irreversible—once a status is set, it should not be allowed to change unless done manually by an administrator. This guarantees transactional consistency and prevents accidental overwrites. A timestamp of the status change should also be recorded to support future reconciliation.

The system should trigger frontend responses accordingly—success pages, receipts, and thank-you messages for successful payments; and retry options or support links for failed payments. The backend must also ensure that duplicate payments are prevented by checking existing payment records for the same order.

**FR7: Save Transaction Data in the Database**

All critical payment information must be permanently stored in the application’s database. This includes razorpay\_order\_id, razorpay\_payment\_id, razorpay\_signature, total amount, status (Paid/Failed), timestamps, and user-related data. These records must be linked to the appropriate order for easy lookup.

This database serves multiple purposes—it acts as the primary transaction ledger for the system, supports reporting and analytics, and provides evidence in case of disputes or refund requests. The system should ensure that this data is written safely and consistently, with constraints to prevent duplication or deletion of payment-critical entries.

For extended functionality, developers may include additional fields such as payment mode, transaction notes, or even Razorpay’s raw API response in JSON format. All sensitive fields must be protected at the model level using Django’s built-in security features.

**FR8: Provide Real-Time Feedback to User**

The system must immediately inform the user of the outcome of their payment attempt—whether it was successful, failed, or cancelled. This response should be clear, visible, and actionable. For successful payments, the user should see a confirmation message or receipt, while for failures, they should be offered a retry option or directed to support.

Feedback mechanisms should include visual indicators (green checkmarks or red alerts), modal popups, redirection to status pages, and possibly email or SMS notifications. This immediate confirmation enhances user trust and ensures they are not left wondering whether their money was received.

Delays in response should be avoided wherever possible. If there is a temporary error during backend verification, a message should advise the user to wait or try again later. All frontend feedback must be consistent with the server’s response to avoid confusion.

**FR9: Allow Admin to Review Payments in Dashboard**

Authorized administrators should be able to view all payment records via Django’s admin interface or a custom dashboard. They must be able to filter orders by status (e.g., Paid, Failed, Pending), search by user or date, and inspect transaction details including payment IDs, timestamps, and signature status.

This functionality supports operational monitoring, fraud detection, refunds, and customer support. Admins should also be able to export transaction reports for accounting or compliance purposes. Future versions of the system may include graphical dashboards showing payment volume trends or user activity.

Access to this feature must be restricted using Django’s admin groups and permissions to prevent unauthorized viewing or modification of financial data.

**FR10: Allow System to Operate in Test Mode (Developer Sandbox)**

During development and testing, the system must support Razorpay’s test mode using sandbox credentials. This allows developers to simulate payments using test cards, UPI IDs, and wallets without involving real money. The system should distinguish between test and live environments to prevent accidental exposure of test transactions in production.

All Razorpay API keys must be securely stored using Django’s settings or environment variables, and never hard-coded. The test mode must include simulated success and failure cases, enabling developers to validate all edge conditions, including retries, timeouts, and cancellations.

**1.4 Non-Functional Requirements**

While functional requirements describe what a system must do, non-functional requirements (NFRs) define **how well** the system performs these functions. They are the backbone of the system’s quality attributes and directly impact its **usability, performance, reliability, security, and maintainability**.

In the case of this project, where secure financial transactions are involved, non-functional requirements play an especially critical role. A failure in these areas could lead to **loss of trust**, **security breaches**, or **system downtime**—all of which are unacceptable in a production-level payment application.

Below are the key non-functional requirements for this Payment Gateway Integration system using Django and Razorpay:

**NFR1: Secure Data Transmission over HTTPS**

    All communication between the client (user browser) and the server must be securely encrypted using HTTPS. This ensures that sensitive data such as order identifiers, Razorpay order details, and user-specific session information is not intercepted or tampered with during transmission.

    The deployment environment should have a valid SSL/TLS certificate installed and properly configured so that every request, especially those involving user authentication or payment, is handled through a secure channel. Django settings should be configured to redirect all HTTP traffic to HTTPS using SECURE\_SSL\_REDIRECT = True and appropriate middleware.

    Using HTTPS not only aligns the system with best practices but is also a requirement for integrating modern payment gateways like Razorpay. Razorpay explicitly denies API calls from non-secure (HTTP) origins in production, making HTTPS integration non-negotiable for live environments.

**NFR2: Cryptographic Signature Verification for Authenticity**

    To ensure the authenticity of each payment, the system must perform server-side signature verification using HMAC SHA256 hashing. This process confirms that the payment details returned to the server—specifically the razorpay\_order\_id, razorpay\_payment\_id, and razorpay\_signature—have not been altered.

    The Razorpay secret key, known only to the backend and Razorpay, is used to generate a signature on the server. This generated signature must be compared to the one sent by Razorpay. Only if the two match is the payment marked as successful and the order status updated.

    This mechanism provides strong security by preventing any tampering or replay attacks, ensuring that each transaction is verifiable, legitimate, and fully traceable. Without this check, malicious actors could falsely mark transactions as complete, leading to major security vulnerabilities.

**NFR3: High Performance and Fast Response Time**

    The system must offer high responsiveness, especially during key interactions like initiating payments, generating orders, and confirming transactions. Page loads, API calls, and database operations should be optimized to ensure that no part of the user journey feels sluggish or unresponsive.

    Requests to Razorpay should be handled asynchronously wherever possible, and Django views must be lean, focusing only on necessary operations per request. For example, payment generation should avoid redundant queries and calculations that can be moved to background jobs or frontend logic.

    A goal of sub-two-second responses on user interactions should be targeted for both mobile and desktop platforms. This helps retain user engagement, especially in time-sensitive environments like food ordering systems during lunch hours or events.

**NFR4: Responsive and Cross-Platform Frontend Design**

    The payment integration must be fully functional across different devices and screen sizes. Razorpay Checkout, embedded on the frontend, should scale correctly whether accessed on a mobile phone, tablet, or desktop computer.

    This responsiveness must be extended to custom payment-related pages as well—such as the cart, order summary, and confirmation screens—ensuring consistency across form fields, images, and buttons. The system should follow responsive design principles using CSS frameworks like Bootstrap, Flexbox, or Tailwind CSS.

    Additionally, compatibility with modern browsers including Chrome, Firefox, Safari, and Microsoft Edge must be ensured. Proper fallback mechanisms should be implemented for devices with JavaScript restrictions or slow connections, thereby ensuring inclusivity in all usage scenarios.

**NFR5: Fail-Safe and Error-Tolerant Payment Handling**

    The system must be designed to tolerate operational errors without crashing or compromising data integrity. When a payment fails—whether due to network errors, user cancellations, or Razorpay API issues—the order should not be marked as paid. Instead, the system should retain the payment attempt, label it as failed, and notify the user gracefully.

    All exceptions must be logged with sufficient detail for debugging, but sensitive information like Razorpay keys or tokens should never be exposed in error messages. Clear and informative feedback must be provided to the user, guiding them to either retry the payment or contact support.

    Graceful degradation ensures that even under unexpected conditions, the system maintains usability and preserves user trust. This is especially important in high-pressure environments where users rely on prompt and reliable payment feedback.

**NFR6: Secure Handling of API Keys and Credentials**

    All sensitive credentials such as the Razorpay KEY\_ID and KEY\_SECRET must be stored securely using Django’s settings file or environment variables. They should not be hardcoded into the source code, JavaScript files, or exposed through HTML templates.

    The .env file containing these credentials should be excluded from version control using .gitignore, and access to it must be restricted to authorized developers or deployment personnel. Key rotation procedures should be in place in case of accidental exposure or suspected compromise.

    Security around key management is foundational to the system’s integrity. Misuse or leakage of these keys could allow attackers to generate fake payment requests or access sensitive transaction data, severely compromising the application and its users.

**NFR7: Scalability and Modularity for Future Enhancements**

    The architecture of the payment system must be modular, separating core functionalities into distinct components such as order creation, payment initiation, verification, and status updating. This separation promotes scalability, allowing the addition of new features like coupons, discount handling, and payment history without disrupting existing logic.

    Code reusability must be emphasized by using helper functions, utility modules, and Django class-based views (if introduced later) for repetitive tasks. The Razorpay integration logic should be encapsulated and reusable for other models or apps within the same Django project.

    This modular approach ensures that the system can scale smoothly when new requirements arise, such as integrating EMI options, subscriptions, or migrating to another payment gateway if required. It reduces technical debt and makes the system future-proof for institutional or commercial adoption.

**NFR8: Maintainability and Code Quality**

    The application must be maintainable by other developers without extensive onboarding. This includes following Django’s project structure, adhering to naming conventions, commenting code blocks where necessary, and organizing templates, static files, and models in a readable hierarchy.

    Code should follow PEP8 guidelines wherever possible, and views should be kept concise with business logic delegated to models or utility functions. All forms, validations, and exception handling must be explicit and well-documented.

    Using version control with meaningful commit messages, along with optional documentation tools like Sphinx or README.md files, will further aid in long-term maintainability. This is especially important if the system is to be handed over to future development teams or scaled as part of a larger institutional platform.

**Chapter 2**

### ****Feasibility Study****

Before embarking on the development of any software system, it is essential to assess whether the project is viable from various perspectives. This assessment is referred to as a feasibility study. The goal of a feasibility study is to evaluate whether the proposed system is practical, affordable, technically achievable, and aligned with user expectations and business goals. For the Payment Gateway Integration project, this study helps in identifying risks, estimating required resources, and ensuring that the system will deliver value to both users and the institution.

    The feasibility of this project is evaluated across four primary dimensions: **Technical Feasibility**, **Economic Feasibility**, **Operational Feasibility**, and **Legal and Social Feasibility**. Each of these aspects provides insight into the likelihood of successful project implementation and long-term sustainability.

**2.1 Technical Feasibility**

    Technical feasibility assesses whether the technology and infrastructure required for the project are available, reliable, and compatible with the system’s goals. In this project, Django has been chosen as the backend framework due to its robustness, modular structure, and built-in security features. Python, being one of the most versatile programming languages, further simplifies the integration of third-party APIs such as Razorpay, reducing development time and complexity.

    Razorpay provides comprehensive developer support through detailed documentation, test credentials, error response guides, and SDKs in multiple languages, including Python. This makes it ideal for integration into Django-based systems. Moreover, all frontend functionalities, including rendering the checkout interface and handling responses, are supported through JavaScript SDKs and secure API calls, which are fully compatible with modern browsers.

    The developers also have access to adequate tools and libraries such as Postman (for API testing), GitHub (for version control), and environments like VS Code or PyCharm, which streamline the development and debugging process. Additionally, the database structure already in place for the CanKIET project can easily accommodate payment-related fields, requiring only minimal schema enhancements. These factors together make the project **technically sound and executable without requiring any new infrastructure or advanced technical training.**

### ****2.2 Economic Feasibility****

Economic feasibility evaluates whether the project is financially viable in terms of the resources required and the benefits it can deliver. One of the major advantages of this project is that it leverages **open-source technologies**, such as Django and Python, which do not incur licensing costs. The Razorpay platform also offers a **free test environment**, allowing developers to implement and fully test the system without any initial investment. Only when the project moves to production and starts processing real payments does Razorpay apply minimal transaction-based charges.

    For a college or small enterprise, the economic burden of maintaining manual payment records, dealing with transaction disputes, and spending time reconciling cash payments can be significantly reduced through automation. This project aims to eliminate such inefficiencies, potentially saving hours of staff time and minimizing errors associated with human data entry. Furthermore, digital payments reduce the need for paper-based receipts and physical handling of cash, indirectly saving costs on stationery and cash management.

    In addition to direct operational savings, the system has the potential to increase **user satisfaction and throughput**, leading to more consistent order volumes and potentially higher revenue for canteen or service providers. In future iterations, if premium features or integrations are introduced (e.g., subscriptions, analytics dashboards), they can be offered as paid add-ons to recover development costs. Overall, the project presents an **excellent return on investment** even if deployed on a small scale.

### ****2.3 Operational Feasibility****

Operational feasibility focuses on how well the new system fits into current workflows and how effectively it will be used by its intended audience. The proposed payment gateway system is specifically designed to complement existing canteen ordering systems by introducing a secure, fast, and intuitive way to complete payments online. This means that the core processes—like item selection, cart management, and order placement—remain familiar to users, while the payment phase is simply streamlined and digitized.

    Since most users in a college setting are already acquainted with digital wallets, UPI, and online payments through services like Paytm or Google Pay, adopting this system will not require significant training or orientation. In fact, the Razorpay Checkout interface closely mirrors the user experience provided by these mainstream apps, making the learning curve practically negligible. The intuitive layout, error handling, and real-time feedback mechanisms further enhance usability and trust.

    On the staff side, administrators and operators will find it easy to monitor and manage transaction records through the existing Django admin interface or a custom dashboard. Payments can be tracked, filtered, and cross-referenced with order IDs, improving accountability and reducing the scope for fraud or mismatch. Integration into day-to-day operations is expected to be seamless and result in better workflow management, faster service delivery, and improved customer satisfaction. The system is, therefore, **operationally practical and scalable within institutional settings.**

### ****2.4. Legal Feasibility****

Legal and social feasibility is essential to ensure that the system operates within regulatory boundaries while being acceptable to its user base. From a legal standpoint, any software dealing with financial transactions must ensure that data security and privacy standards are upheld. Fortunately, the Razorpay platform is **PCI-DSS compliant**, meaning that all sensitive payment information (such as card numbers and CVVs) is securely managed by Razorpay’s infrastructure, and not by the Django application itself.

    The system must also comply with the provisions outlined in India’s **Information Technology Act, 2000**, especially regarding electronic transactions and the secure handling of personal data. To remain compliant, the Django backend must ensure secure storage of only non-sensitive metadata (e.g., order ID, status, payment ID) and should provide privacy notices to users if additional data is collected.

    Socially, the platform aligns with modern expectations for digital convenience. Particularly in the context of a post-pandemic world, there is a heightened demand for **contactless transactions** and reduced dependence on physical cash. Students, faculty, and staff already rely on smartphones and apps for various academic and personal tasks, making this integration a natural progression. Furthermore, digital payments bring a sense of transparency and reliability that can foster trust in institutional transactions.

    Considering these legal safeguards and the alignment with user habits and societal trends, the system is both **legally sound and socially well-positioned** for implementation.

### ****2.5 Scheduling Feasibility****

   Scheduling feasibility focuses on assessing whether the system can be designed, developed, tested, and deployed within the time constraints imposed by project requirements, academic deadlines, or institutional goals. It considers both the planned development timeline and the availability of necessary resources to meet these deadlines without compromising on quality or functionality.

    In the context of the Payment Gateway Integration project, the timeline has been strategically structured into clearly defined phases, each with specific objectives and deliverables. The project follows a linear, milestone-driven approach to ensure that progress can be tracked efficiently, and any delays can be addressed promptly through adjustments in workload distribution or resource allocation.

    The following is a breakdown of the proposed timeline:

* **Requirements Gathering (2 weeks):**  
      During this phase, functional and non-functional requirements are collected through brainstorming sessions, research on Razorpay API documentation, and discussions with stakeholders. Detailed study of the Django backend system and its existing models, particularly the Order and Items models, is conducted to assess integration needs.
* **System Design (2 weeks):**  
      In this stage, the architecture of the system is planned. Key components such as order creation, Razorpay API communication, frontend rendering of the checkout interface, and backend signature verification are modeled. Wireframes for the payment interface and flowcharts for backend logic are created to visualize the system.
* **Development and Testing (6 weeks):**  
      This is the most resource-intensive phase, covering coding, Razorpay API integration, database updates, form handling, view management, JavaScript checkout logic, and rigorous unit and integration testing. Extensive testing is carried out in Razorpay’s test environment to simulate real-world scenarios, including payment success, failure, and signature mismatches. Edge case handling and user interface refinements are also addressed.
* **Deployment and Final Review (1 week):**  
      The application is deployed in a controlled environment using Django's production settings, and real Razorpay keys are integrated. Post-deployment testing ensures that the system is secure, stable, and user-friendly. Final documentation is prepared, and a demonstration is scheduled for evaluation or handover.

    In terms of **resource availability**, the project is supported by a development team with working knowledge of Django, frontend web development, and API handling. All essential tools—such as code editors, version control platforms (e.g., GitHub), test servers, and Razorpay’s sandbox account—are already accessible. Moreover, having prior experience from the CanKIET project allows the team to reuse certain structures and focus more efficiently on the payment functionality.

    Considering the clarity of deliverables in each stage, the familiarity with tools and frameworks, and the logical division of tasks, the project is deemed **highly feasible from a scheduling perspective**. Adequate buffer time is also included to handle unexpected challenges, ensuring timely completion without compromising the system's robustness.

### ****Chapter 3****

### ****Project Objective****

    The objective of the **Payment Gateway Integration** project is to develop a secure, scalable, and user-friendly payment system that enables online transactions within a web application environment, specifically within the context of an institutional or organizational setting. The project aims to streamline the payment process for users—students, staff, or customers—by integrating the **Razorpay payment gateway** into an existing Django-based platform. By enabling digital payments, the system reduces dependency on physical cash transactions and manual accounting, while ensuring fast, reliable, and verifiable payment processing.

    In today’s increasingly digitized world, users expect seamless, secure, and real-time transaction capabilities within web applications. This project addresses that expectation by providing a robust solution that allows users to view their orders, confirm their purchases, and make payments through widely accepted digital payment methods such as UPI, credit/debit cards, net banking, and wallets. At the same time, the system ensures that every payment is validated using cryptographic verification to prevent fraud and maintain transactional integrity.

    The key goal is to implement a **full payment workflow**, starting from order initiation to successful transaction confirmation, including order creation at the backend, Razorpay checkout integration on the frontend, and post-payment verification logic. Additionally, the project introduces server-side data recording for payment status, payment ID, and signatures to ensure auditability and record-keeping. This backend integration provides administrators with real-time insight into transactions and order statuses.

    Another core objective is to deliver a **developer-friendly and maintainable** architecture. The integration must follow Django best practices, maintain modularity, and include clear documentation and comments for future enhancements, such as support for subscriptions, webhooks, refunds, or recurring billing. The system should also allow flexibility in adapting the Razorpay integration for different use cases across other institutional applications or business models.

    Ultimately, this project aims to combine **technical reliability, financial security, and user-centric design** to build a real-world, production-ready payment integration system that aligns with current market needs and can evolve with future technological advancements.

    The **Payment Gateway Integration** project is a crucial step toward modernizing the ordering and transaction experience within web applications like the CanKIET platform. By embedding a reliable, real-time digital payment workflow, the system minimizes dependency on cash, reduces transaction friction, and creates a more professional, secure experience for users and operators alike. The integration is not just a technical add-on—it is a foundational enhancement that touches every aspect of the ordering lifecycle.

    The project aims to address key gaps in traditional payment processing by leveraging Razorpay’s comprehensive API offerings within Django's robust backend framework. It introduces flexibility, auditability, and automation into what was once a largely manual process, thereby increasing system efficiency and trust. Additionally, the integration ensures proper storage of transactional records and supports future extensions such as refunds, invoice generation, and user-based payment histories.

    The **Payment Gateway Integration** system aims to achieve the following key objectives:

1. **Enable Secure Digital Transactions:**  
       Implement a robust and trusted payment system using Razorpay that allows users to securely pay through UPI, cards, wallets, or net banking. This objective reduces cash handling, speeds up checkout times, and ensures transaction safety through server-side verification and encrypted communication protocols.
2. **Integrate Razorpay API Seamlessly into Django:**  
       Utilize Razorpay’s Python SDK to create server-side orders, initiate the checkout process, and verify payment signatures. Ensure that all components work harmoniously with Django’s views, models, and templates while adhering to the MVC architectural pattern.
3. **Ensure Real-Time Order and Payment Linking:**  
       Connect payment information directly with the order model by linking Razorpay’s order\_id, payment\_id, and verification signature to internal order records. This allows users and administrators to track payment success or failure instantly with clear backend visibility.
4. **Enhance the User Checkout Experience:**  
       Provide a clean and intuitive frontend experience that includes Razorpay’s embedded checkout form. Ensure users can initiate payment with a single click, view transaction status updates, and get redirected or notified upon success or failure—all without confusion or delay.
5. **Minimize Payment Failures and Errors:**  
       Implement proper exception handling and feedback mechanisms so that even if a payment fails, the system gracefully informs the user and logs the failure for review. This objective builds user trust by preventing frustration and uncertainty during the transaction.
6. **Store and Manage Transaction Metadata Securely:**  
       Store non-sensitive but important payment-related metadata such as transaction status, timestamps, user ID, amount paid, and payment gateway response fields. This information is vital for audits, generating receipts, and resolving disputes if needed.
7. **Support Scalability and Modular Design:**  
       Build the payment module in a modular manner so it can be reused across other parts of the system or integrated into other Django-based platforms in the future. Ensure that core logic like Razorpay order creation, signature verification, and response handling are encapsulated for reusability.
8. **Comply with Legal and Financial Guidelines:**  
       Ensure that all transactions are conducted in accordance with industry standards for digital payments. Avoid storing sensitive card details, use HTTPS for all endpoints, and follow Razorpay’s guidelines for safe API use and transaction handling.
9. **Integrate Payment Status into Admin Dashboard:**  
       Allow canteen or business administrators to view which orders are paid, pending, or failed directly from the backend. This helps in efficient service delivery and allows tracking customer fulfillment in real time.

**Chapter 4**

#### **Hardware and Software Requirements**

    For the development, integration, testing, and deployment of the **Payment Gateway Integration** system using Django and Razorpay, a precise and well-documented set of hardware and software specifications is essential. These requirements ensure that the application functions reliably across different environments and is scalable for future use in production-level systems. Since the application involves real-time payment handling, secure API communication, and database transaction processing, the underlying hardware and software must be robust enough to support these operations without delays or disruptions.

    This chapter outlines both **minimum and recommended configurations** required to develop and deploy the system effectively. It includes requirements for the developer machine, staging environment, and optional hosting server (if applicable). Additionally, it covers dependencies, tools, and services necessary for Razorpay API interaction and Django compatibility. Meeting these requirements allows for a seamless integration process, minimizes development errors, and ensures that the system can handle real-time payment workflows with high reliability.

**4.1 Hardware Requirements**

    Hardware requirements play a vital role in determining the performance and responsiveness of the development environment. Inadequate hardware resources can lead to system lags, longer build times, and inefficient debugging cycles. On the other hand, well-equipped systems allow developers to code, test, and simulate payment transactions more efficiently, especially when working with real-time API calls and database operations.

**Minimum Hardware Requirements (for Development & Local Testing):**

* **Processor:** Dual-core 2.0 GHz or higher  
      Sufficient for running Django’s development server and basic debugging tools. However, parallel processes like browser-based testing and IDE operations may slow down performance.
* **RAM:** 4 GB  
      Minimum to run IDEs like VS Code or PyCharm alongside a web browser. Suitable for lightweight Django projects but may not handle multiple tabs or heavy libraries efficiently.
* **Hard Disk:** 50 GB of free disk space  
      Needed to accommodate Python packages, IDEs, datasets, and temporary files. Disk space is also important when installing system-wide dependencies and local databases.
* **Monitor:** 1024x768 resolution display  
      Sufficient for basic UI layout testing but may limit productivity in design tasks or responsive layout validation.
* **Input Devices:** Standard keyboard and mouse  
      Essential for interaction with development tools and browser-based testing.
* **Internet Connection:** Required  
      Necessary for installing packages, accessing Razorpay API services, and retrieving documentation. Must be stable for real-time testing.

**Recommended Hardware Requirements (for Optimal Development Experience):**

* **Processor:** Intel Core i5 (8th Gen or newer) or AMD Ryzen 5 equivalent  
      Enables smooth multitasking, compilation, real-time debugging, and testing across browsers simultaneously.
* **RAM:** 8 GB or more  
      Recommended for running multiple services in parallel (e.g., database server, Django server, frontend tools, Git clients).
* **Hard Disk:** 100 GB SSD  
      Solid-state drives offer faster I/O operations, significantly reducing application load and test cycles.
* **Monitor:** Full HD (1920x1080) display  
      Allows better UI design testing and multiple side-by-side windows (e.g., browser, editor, terminal).
* **Internet Connection:** Broadband (10 Mbps or higher)  
      Critical for sending/receiving API calls, loading SDKs, testing Razorpay in live or sandbox mode.

    For **cloud-based hosting**, a virtual private server (VPS) or a cloud instance (like AWS EC2 or DigitalOcean droplet) should have at least 2 GB RAM, 1 vCPU, and 25 GB SSD for lightweight deployment. Production deployments should scale resources based on concurrent transaction volume and expected traffic.

**4.2 Software Requirements**

    Software requirements define the platforms, programming languages, frameworks, libraries, and utilities that power the application. Since this project is built using the Django framework and integrates a third-party API (Razorpay), the environment must support Python-based development, secure networking, and JavaScript-based frontend interactions. Furthermore, development tools such as version control systems, virtual environments, and debugging utilities are necessary to ensure maintainable and scalable application delivery.

**Operating System:**

* **Development OS:**  
      - Windows 10/11  
      - Ubuntu 20.04+  
      - macOS Monterey or later  
      All are compatible with Python and Django. Linux (Ubuntu) is recommended for better compatibility with deployment scripts and server behavior.
* **Deployment OS:**  
      - Ubuntu 20.04/22.04 LTS (server edition)  
      Linux servers are preferred for hosting Django applications due to performance, scalability, and better package control.

**Backend Stack:**

* **Programming Language:** Python 3.8 or newer  
      Python provides compatibility with Django 4.x and the Razorpay SDK. The use of venv for virtual environments ensures dependency isolation.
* **Web Framework:** Django 4.x  
      Includes built-in security features, ORM for model interaction, and support for custom views for Razorpay handling.
* **Package Installer:** pip  
      Used to install all required third-party packages.
* **Database:**  
      - **Development:** SQLite (lightweight, file-based, included by default)  
      - **Production (optional):** PostgreSQL or MySQL for scalability and performance with concurrent users.
* **Server (Production):** Gunicorn + Nginx (recommended)  
      For handling concurrent requests securely and efficiently.

**Frontend Technologies:**

* **Languages:** HTML5, CSS3, JavaScript  
      Used for rendering the UI, integrating Razorpay’s checkout widget, and handling user interactions.
* **Libraries/Frameworks:**  
      - Bootstrap 5 (for responsive layout)  
      - Razorpay Checkout.js (provided by Razorpay to render the payment form)  
      JavaScript is crucial for rendering the payment modal, managing callbacks, and capturing Razorpay responses.

**Key Python Packages/Dependencies:**

* razorpay: Razorpay’s official SDK for Python integration
* requests: For handling custom API calls if required
* python-dotenv: For managing environment variables and securing API keys
* django-crispy-forms: Optional, for rendering better styled forms
* django-environ: Optional, for environment variable-based configuration management

**Development Tools:**

* **IDE/Text Editor:** Visual Studio Code, PyCharm, or Sublime Text  
      VS Code is lightweight and extensible. PyCharm provides deep Django support and database management integration.
* **Version Control:** Git with GitHub  
      Used to manage source code versions, collaborate with team members, and maintain project history.
* **API Testing:** Postman  
      Useful for testing API endpoints such as Razorpay order creation, signature verification, and response validation.
* **Terminal/Command Line:**  
      Bash (Linux/macOS) or Command Prompt/PowerShell (Windows) for managing migrations, server run commands, and environment configurations.

**Razorpay Development Prerequisites:**

* **Razorpay Test API Keys** for development  
      Enables creation of test orders and simulation of successful or failed transactions.
* **Razorpay Live Keys** for production  
      To process real payments once testing is complete and the system is launched.
* **Callback/Redirect URLs** must be allowlisted in the Razorpay dashboard to ensure successful signature verification and response redirection.

## **API Calling Using API Keys for Payment Gateway Integration**

An **API (Application Programming Interface)** is a set of protocols and tools that enables software components to communicate with each other. In the context of payment gateways, APIs act as the bridge between the merchant’s website (or application) and the payment service provider.

When a user initiates a payment, the website communicates with the payment gateway’s system via secure API calls. These API calls ensure that payment information (such as amount, currency, and customer details) is properly transmitted, validated, processed, and responded to in real time.

Payment gateways like **Razorpay, PayPal, Stripe, and Paytm** provide RESTful APIs that use HTTP methods (like POST, GET) to initiate transactions, fetch payment statuses, create orders, verify payments, and more.

### ****Understanding API Keys****

To ensure secure communication between the application and the payment gateway, **API keys** are used. These keys are generated by the payment gateway provider and are unique to each merchant.

There are typically two types of keys:

* **Public Key (Key ID)**: This key is used on the client side (frontend). It identifies the merchant account but does not allow unauthorized access.
* **Private Key (Key Secret)**: This key is stored securely on the server side and is used to sign or verify transactions, and to authorize actions like order creation, refunds, and payment verification.

The API keys act like a **username and password**, and must be handled securely to prevent any unauthorized access or misuse.

### ****3. API Workflow in Payment Integration****

Here’s a step-by-step breakdown of how APIs work in a typical payment gateway integration:

#### ****Step 1: Key Generation and Authentication Setup****

* The developer logs into the payment gateway dashboard.
* The platform provides a **test mode** and a **live mode**, each with separate API keys.
* These keys are securely integrated into the application’s backend configuration.

#### ****Step 2: Order Initialization****

* When a user clicks "Pay Now", the application sends a **secure request** to the payment gateway to create a new order.
* The request includes:
  + Transaction amount
  + Currency type
  + Receipt ID or Order ID
  + Customer details
* The gateway responds with a **unique Order ID**, which is essential to link the transaction.

#### ****Step 3: Checkout and Payment Form****

* The frontend uses the **Order ID and Key ID** to open a **payment form** provided by the gateway (e.g., Razorpay Checkout).
* The customer enters their payment details and confirms the transaction.

#### ****Step 4: Payment Processing****

* Once the customer confirms payment, the gateway:
  + Processes the transaction through banks or UPI/Wallets.
  + Returns a response to the frontend with payment details such as:
    - Payment ID
    - Order ID
    - Digital Signature

#### ****Step 5: Signature Verification and Validation****

* The backend now takes these details and verifies their authenticity using the **Key Secret**.
* This verification process ensures that the payment was not tampered with.
* If verification is successful, the order status is marked as "Paid" in the application.

#### ****Step 6: Transaction Completion****

* After successful verification:
  + The customer receives a confirmation message or invoice.
  + The merchant receives the amount in their settlement account as per the gateway's settlement cycle.

### ****4. Importance of Signature Verification****

The most critical part of secure payment integration is **verifying the digital signature** sent by the gateway after a transaction. This signature proves that:

* The payment was initiated by the actual gateway.
* The payment was not intercepted or altered.
* The transaction data is consistent and validated.

Without signature verification, the system would be vulnerable to **fraudulent payment confirmations**, leading to false order statuses.

### ****5. Advantages of Using APIs with API Keys****

| **Benefit** | **Explanation** |
| --- | --- |
| **Security** | API keys ensure only verified apps can make payment requests. |
| **Automation** | Payment processes (creation, confirmation, refunds) are automated through API calls. |
| **Scalability** | APIs allow the application to handle thousands of payments efficiently. |
| **Real-time Data Access** | APIs provide instant updates on payment success/failure. |
| **Flexibility and Customization** | Developers can design custom workflows using gateway APIs. |
| **Test Mode Availability** | APIs support a test environment to simulate payments before going live. |

### ****6. API Communication and Data Flow****

The data communication between the application and payment gateway using API calls is **bidirectional** and **secure**. The overall flow is:

1. **App to Gateway (Request):**
   * Order creation request is made with parameters such as amount, customer, and description.
   * The request is authenticated using the Key Secret.
2. **Gateway to App (Response):**
   * An order ID or response object is sent back to the app.
   * This response is then used to initiate the customer-facing checkout interface.
3. **Customer to Gateway (Payment):**
   * The customer enters card/UPI details securely on the gateway's form.
4. **Gateway to App (Verification):**
   * On successful payment, the gateway sends a callback to the backend with payment data and a signature.
   * The backend verifies this data before confirming order status.

### ****7. Common Use Cases of Payment Gateway APIs****

| **API Function** | **Use Case** |
| --- | --- |
| Order Creation | Before initiating a payment session |
| Payment Verification | After transaction is complete |
| Refund Initiation | Issuing partial/full refunds |
| Fetching Payment Status | Checking real-time payment outcomes |
| Webhook Notifications | Receiving payment event alerts |

### ****8. Best Practices for Using Payment Gateway APIs****

* **Never expose the Key Secret in frontend code.**
* **Always use HTTPS** for API communication to encrypt data in transit.
* **Store sensitive keys in environment variables**, not hardcoded in files.
* **Use webhook validations** to secure automatic transaction updates.
* **Rotate API keys periodically** to minimize the risk of leaks.
* **Log API responses** (excluding card details) for audit and debugging purposes.

### ****9. Conclusion****

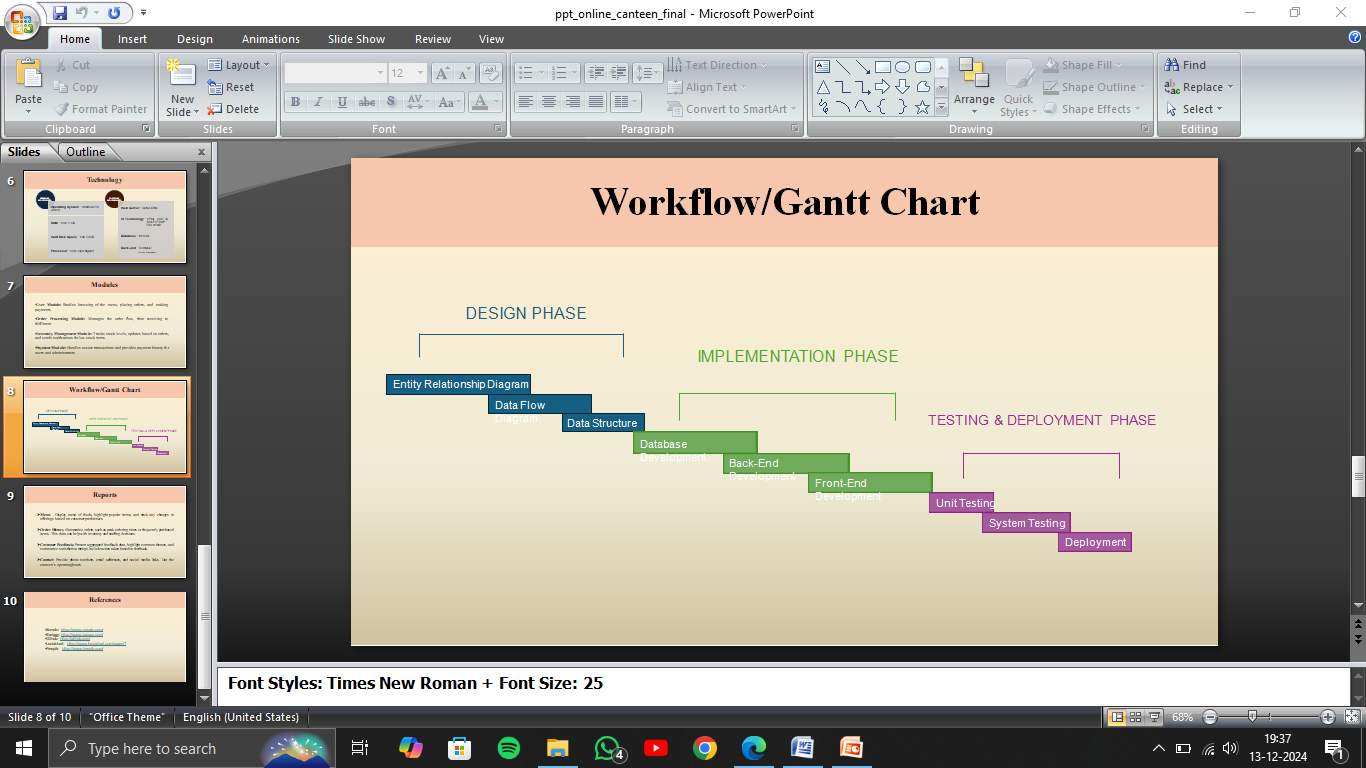
The use of APIs and API keys is at the heart of modern, secure, and reliable payment gateway integration. These APIs allow seamless communication between a web application and the payment service provider, ensuring fast, accurate, and secure transactions. By understanding the working of API keys, order creation, signature verification, and response handling, developers can build robust payment systems that offer a smooth experience to users while protecting against fraud and data compromise.

#### ****Chapter 5****

#### ****5.1 Project Flow****

    The development of the **Payment Gateway Integration** system follows an agile and modular approach, allowing for iterative development and frequent validation. This methodology ensures that each component—ranging from order creation to payment verification—is built, tested, and refined in stages for accuracy, efficiency, and maintainability.

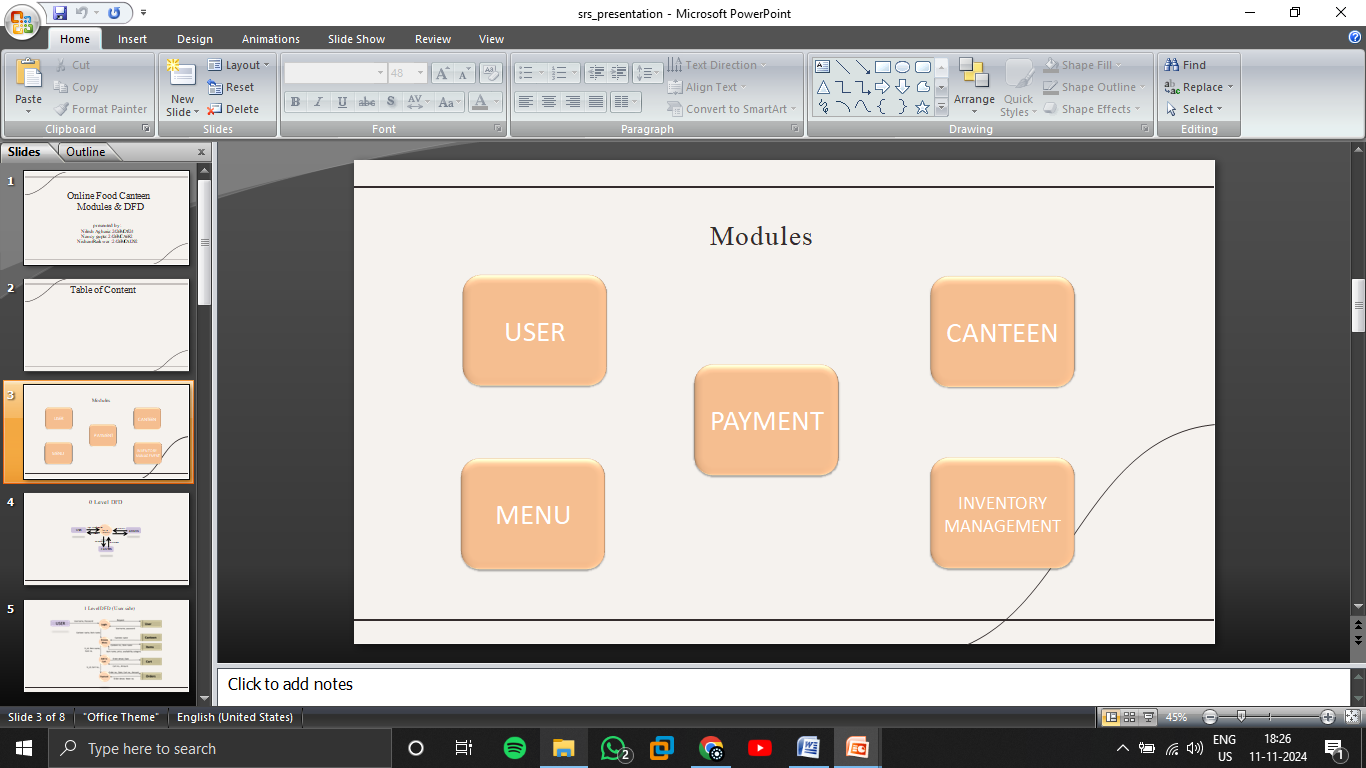
    The flow of development includes requirement gathering, system design, Razorpay integration, payment status verification, and transaction storage. The entire cycle is validated through testing and deployed on a secure server environment.



**Project Development Stages (Fig 5.1):**

1. **Requirement Gathering and Analysis:**  
       - Identify the need for a secure, real-time payment system.  
       - Understand Razorpay's API capabilities and constraints.  
       - Study Django models to support order and payment data structure.
2. **System Design:**  
       - Design database schema for Order and Payment integration.  
       - Create backend architecture for secure API calls and payment verification.  
       - Plan user interface flow for initiating payment via Razorpay Checkout.
3. **Implementation:**  
       - Integrate Razorpay order API on the backend using Python SDK.  
       - Render the Razorpay Checkout form on the frontend with necessary options.  
       - Handle POST responses and verify Razorpay signature to confirm authenticity.
4. **Testing:**  
       - Use Razorpay sandbox for successful and failed transaction simulation.  
       - Validate order creation, payment linkage, and signature verification.  
       - Test edge cases like timeouts, incorrect signature, and network failures.
5. **Deployment:**  
       - Deploy on a cloud-based server with HTTPS support.  
       - Configure test/live Razorpay keys securely via environment variables.
6. **Maintenance and Feedback:**  
       - Monitor real payment performance.  
       - Handle user queries related to payment failures or confirmations.  
       - Update system based on Razorpay API version changes or business needs.

**5.2 Modules of Payment Gateway Integration System**

****

**System Modules Overview (Fig 5.2):**  
The system is built on modular principles, with each functional component focusing on a specific responsibility within the payment process.

**1. User Module**

**Description:**  
    Handles user interactions throughout the payment journey, including initiating payments and viewing transaction statuses.

**Features:**

* User Identification (via session or login context)
* Display of selected items and total price
* Initiation of Razorpay Checkout
* Viewing of payment success/failure status

**2. Order Module**

**Description:**  
    Manages creation and storage of order data including item, quantity, and price, and links it to payment transactions.

**Features:**

* Order creation via server-side Razorpay API call
* Association of unique Razorpay Order ID to each transaction
* Tracking of payment status (is\_paid, payment\_id, signature)
* Storing metadata like order time, total cost, and item info

**3. Payment Module**

**Description:**  
    Manages the entire lifecycle of a payment—initiation, processing, confirmation, and verification.

**Features:**

* Razorpay Checkout Form Integration
* Callback or handler function for post-payment processing
* Signature verification using Razorpay Python SDK
* Transaction record update on success/failure
* JSON response handling and redirection after payment

**4. Admin Module**

**Description:**  
    Provides backend control to view payment records, check order statuses, and manage API keys securely.

**Features:**

* View orders filtered by payment status
* Update system configurations (e.g., Razorpay keys, callback URLs)
* Generate transaction logs and summary reports
* Access order-payment linkage data for audit purposes

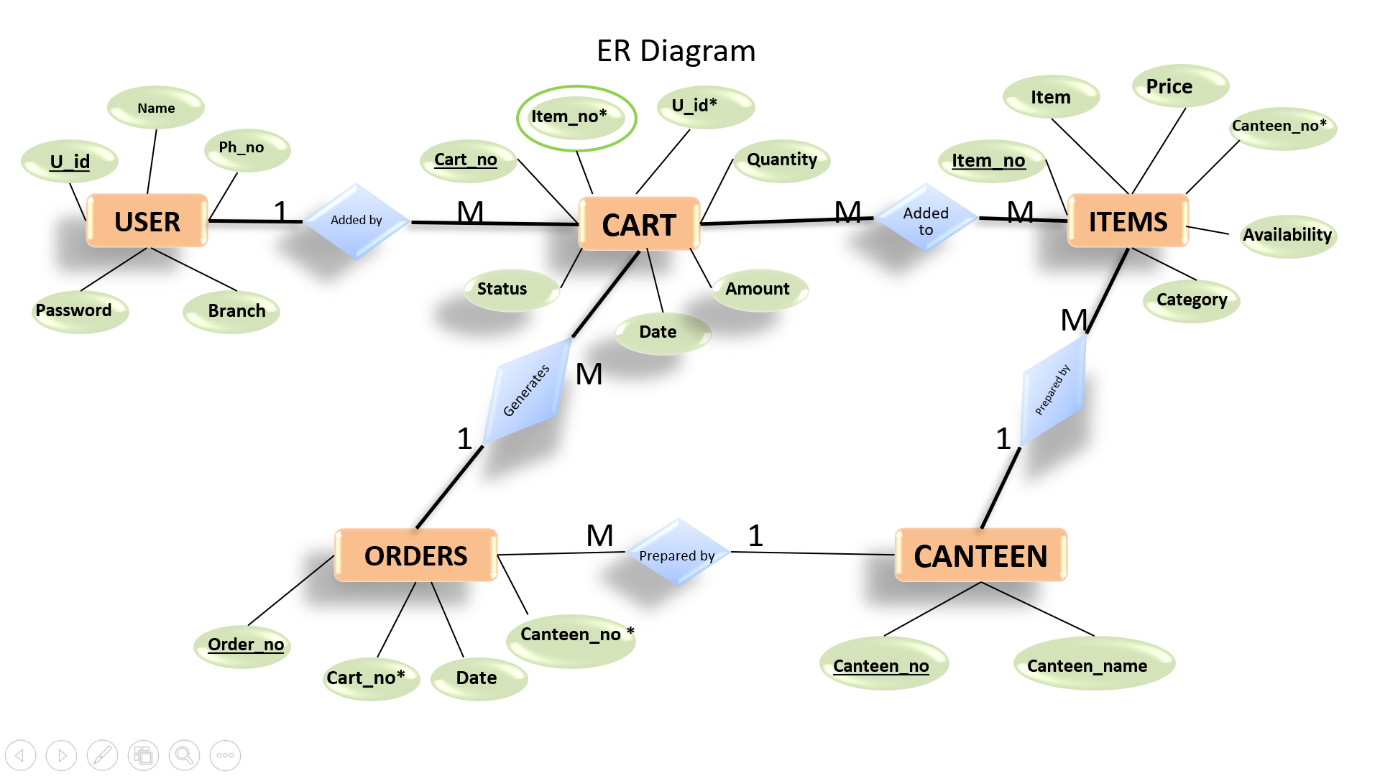
**5. Verification and Security Module**

**Description:**  
    Ensures payment authenticity by verifying the Razorpay signature returned from the checkout.

**Features:**

* HMAC-SHA256 signature generation on the server
* Comparison of generated vs. returned signature
* Error handling for mismatches or invalid responses
* Logging of all verification attempts for security audits

**5.3 ER Diagram**



**Fig 5.3: Entity Relationship Diagram**  
The ER diagram represents the core entities involved in the payment system and their inter-relationships.

**1. Entities and Their Attributes**

**1.1 User Entity**

* **Attributes**:
  + user id (Primary Key)
  + name
  + phone no
  + branch
  + password

**1.2 Cart Entity**

* **Attributes**:
  + cart no (Primary Key)
  + item no (Foreign Key referencing Item)
  + quantity
  + amount
  + date
  + status
* **Relationships**:
  + A User can have many Cart (1-to-many), meaning a user can have multiple carts.
  + A Cart can contain multiple Items, and multiple items can be added to a single cart (many-to-many relation).

**1.3 Item Entity**

* **Attributes**:
  + Item no (Primary Key)
  + item name
  + price
  + canteen no (Foreign Key referencing Canteen)
  + availability
  + category
* **Relationships**:
  + Items are served in a Canteen, so there's a 1-to-many relationship between Canteen and Item (a canteen can serve multiple items).
  + An Item can be added to multiple Carts (many-to-many relationship).

**1.4 Orders Entity**

* **Attributes**:
  + order no (Primary Key)
  + cart no. (Foreign Key referencing Cart)
  + date
* **Relationships**:
  + A Cart can result in multiple Orders (1-to-many), indicating a cart can be processed into one or more orders.

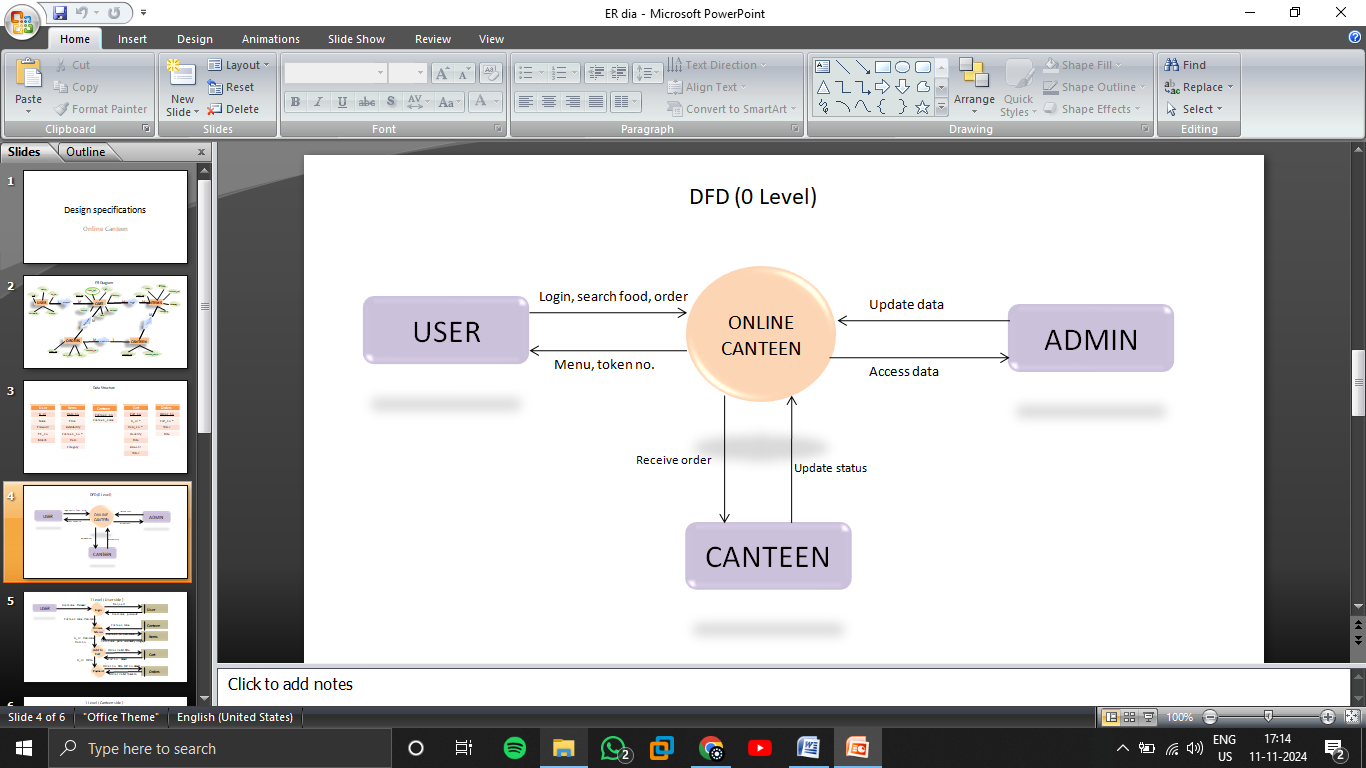
**1.5 Canteen Entity**

* **Attributes**:
  + canteen no. (Primary Key)
  + canteen name
* **Relationships**:
  + A Canteen can serve multiple Items (1-to-many).
  + There’s a relationship between Canteen and Orders (1-to-many), where a canteen can fulfill many orders.

**5.4 Data Flow Diagrams (DFD)**

**0 Level DFD (Context Diagram)**

The **Zero-Level Data Flow Diagram (DFD)** represented in **Fig 5.4.1** is the highest level of abstraction in the system, providing an overview of the core processes and how the system interacts with external entities. For the **Online Canteen** system, the three primary external entities interacting with the system are **User**, **Admin**, and **Canteen**. Below is the description of the Zero-Level DFD for the Online Canteen system.



**Fig 5.4.1: Zero Level DFD**

**Entities:**

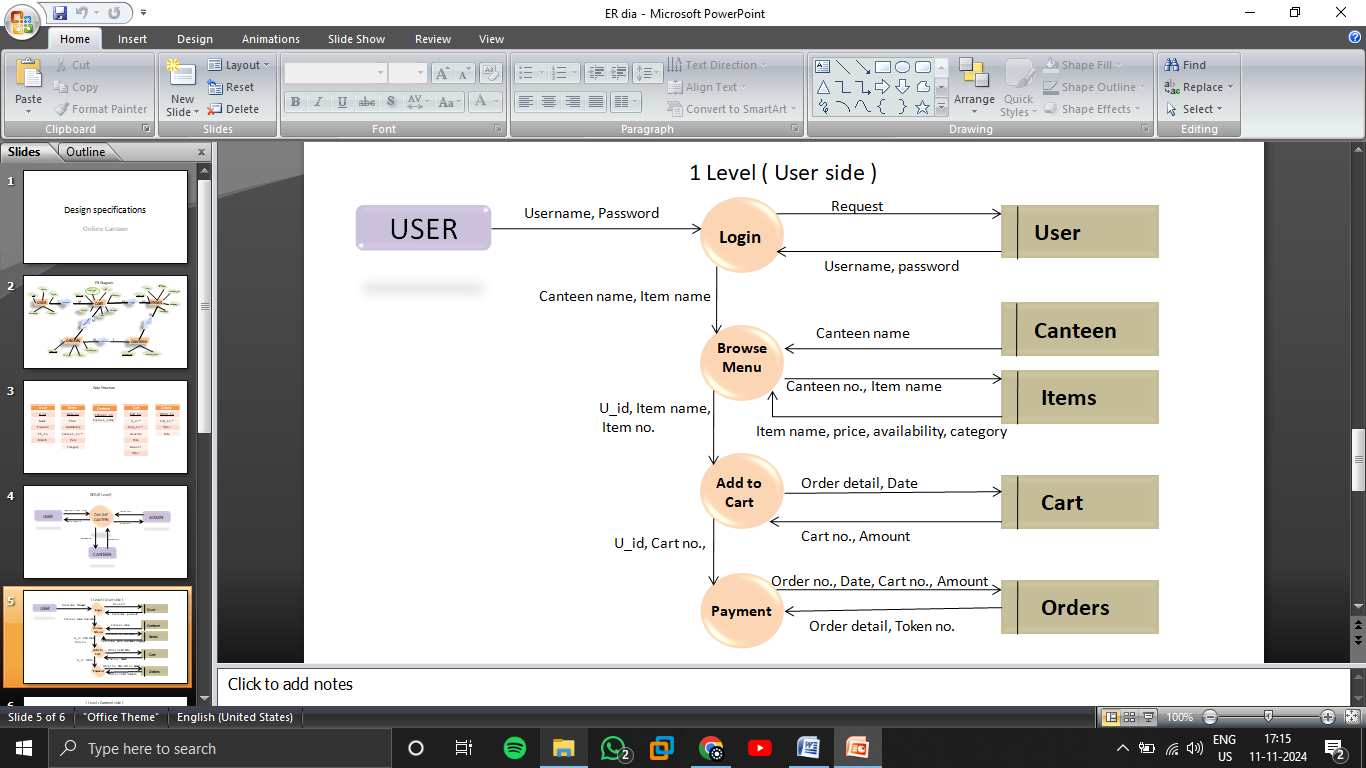
* **User:** Initiates the payment, receives confirmation.
* **Admin:** Monitors orders and payments.
* **Razorpay:** Processes payments and returns responses.

**Process:**

* User places order → System generates Razorpay order → Razorpay processes payment → System verifies and stores result

**Level 1 DFDs**

The **Level 1 Data Flow Diagram (DFD)** for the **Online Canteen System** breaks down the Zero-level DFD into more detailed processes that describe the interactions between the **User ( Fig 5.5 )** entities with the main functional components of the system. This level captures specific functionalities that each entity interacts with, such as ordering, managing the menu, and handling inventory.

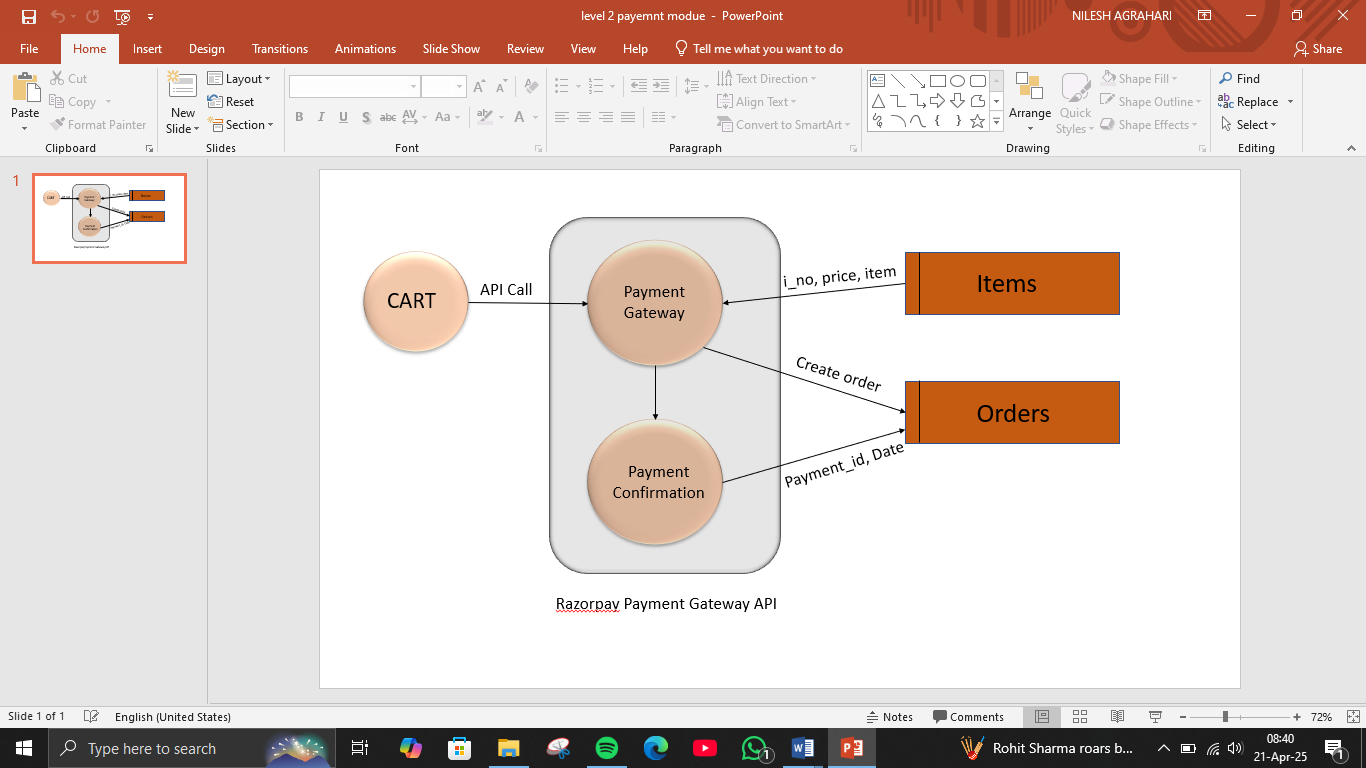


**User Side (Fig 5.5)**

* User selects item → Adds to cart → Proceeds to payment
* System generates Razorpay order → User completes payment
* System verifies payment and updates status → User receives confirmation

**Level 2 Payment API**

The Level 2 DFD for the Payment Gateway Integration focuses on the internal processes involved in handling payment requests, verifying transactions, and updating the system accordingly. This diagram highlights how the system communicates with Razorpay's payment services to process user payments securely and efficiently.

****

**Payment integration (fig 5.6)**

Description of Processes and Flow:

1. **CART (External Entity):**
   * This represents the user's shopping cart that contains selected items.
   * Once the user proceeds to checkout, an API Call is initiated to the Payment Gateway process with necessary data such as item details and total price.
2. **Payment Gateway (Process):**
   * This internal process takes the request from the cart and interacts with the Razorpay Payment Gateway API.
   * It retrieves the item data such as i\_no (item number), price, and item name, referencing the Items data store.
   * It then creates a new order, forwarding the data to the Orders data store.
3. **Payment Confirmation (Process):**
   * After the user completes the payment through Razorpay’s frontend, the confirmation data is captured.
   * This includes payment\_id and Date, which are sent to the Orders data store to update the status of the respective order.
4. **Data Stores:**
   * Items: Stores the list of all items available for purchase, including item number, name, and price.
   * Orders: Maintains details of all created orders, including payment ID, date, and order status.
5. **External Payment API:**
   * The interaction between the Payment Gateway and Razorpay happens through secure API calls, ensuring the transaction is encrypted and verified before marking an order as paid.

**Purpose of This Level 2 DFD:**

This DFD provides a more granular view of the payment module, breaking it down into two main components:

* Payment Gateway (Initiation)
* Payment Confirmation (Verification and Update)

It helps in visualizing how the system ensures secure payment handling, manages the flow of data between cart, Razorpay, and order management, and guarantees consistent synchronization between item data and order confirmation.

#### ****Chapter 6****

#### ****Project Outcome****

The Payment Gateway Integration Project is designed to achieve the following outcomes:

* A robust, fully functional web-based platform that streamlines canteen operations, now enhanced with secure online payment capabilities through integrated Razorpay payment gateway.
* A significantly improved user experience for students and faculty, featuring a seamless and simplified ordering and payment process. Users can easily select items, place orders, and complete transactions online, reducing wait times and manual cash handling.
* Increased operational efficiency for canteen staff, as payment automation reduces manual transaction processing, minimizes errors, and accelerates order fulfilment.
* Real-time transaction data and analytics, empowering canteen management with insights into sales trends, payment statuses, and customer preferences for data-driven decision-making.
* Enhanced satisfaction and convenience for all stakeholders, as the system offers secure, transparent, and user-friendly payment options, leading to higher trust and adoption rates among users.

By integrating a payment gateway into the Django-based canteen platform, the project not only modernizes the payment process but also delivers a comprehensive solution that benefits students, faculty, staff, and management alike.

**Screenshots of final project**

**User interface**

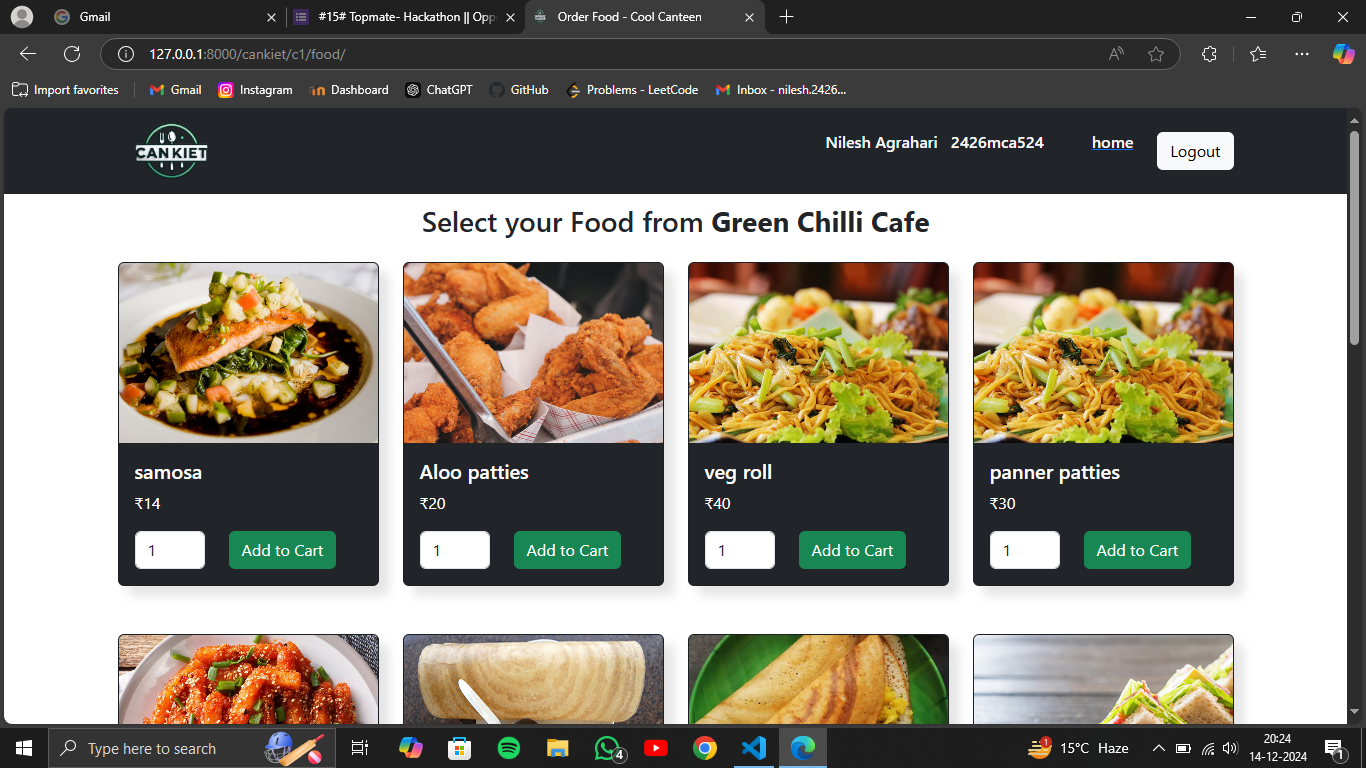


Fig 6.4 Menu Section

The CanKiet item selection page as shown in Figure 6.4 displays a curated selection of food items available from a specific canteen. The page prominently features the name of the selected canteen, allowing users to easily identify the source of the menu items. Each food item is presented with an image, its name, price, and a quantity selector. Users can easily adjust the quantity of each item they wish to order.

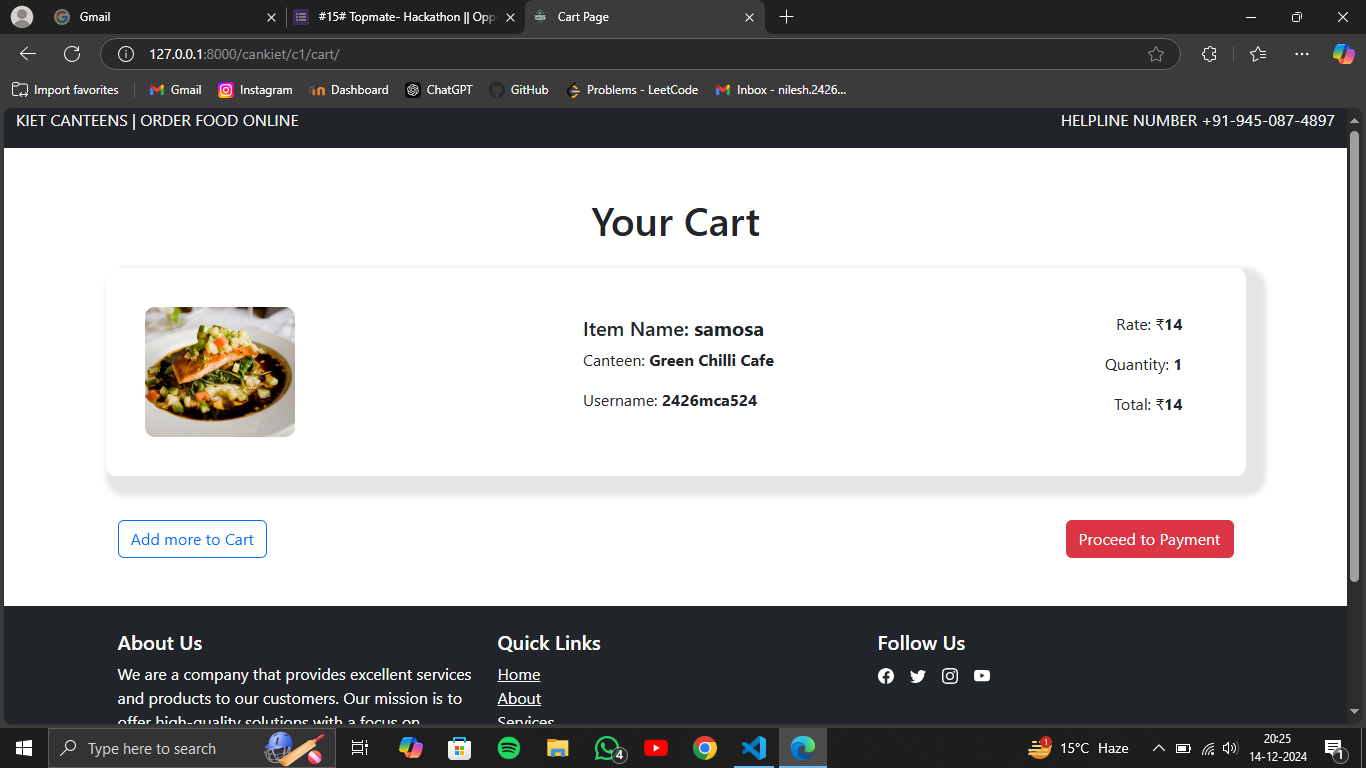


Fig 6.5 Cart page

The CanKiet cart page as shown in Figure 6.5 displays a clear summary of the items a user has added to their order. It includes the name of the item, the canteen it's from, the quantity selected, the price per item, and the total cost. A prominent image of the item is also displayed, providing a visual reference.

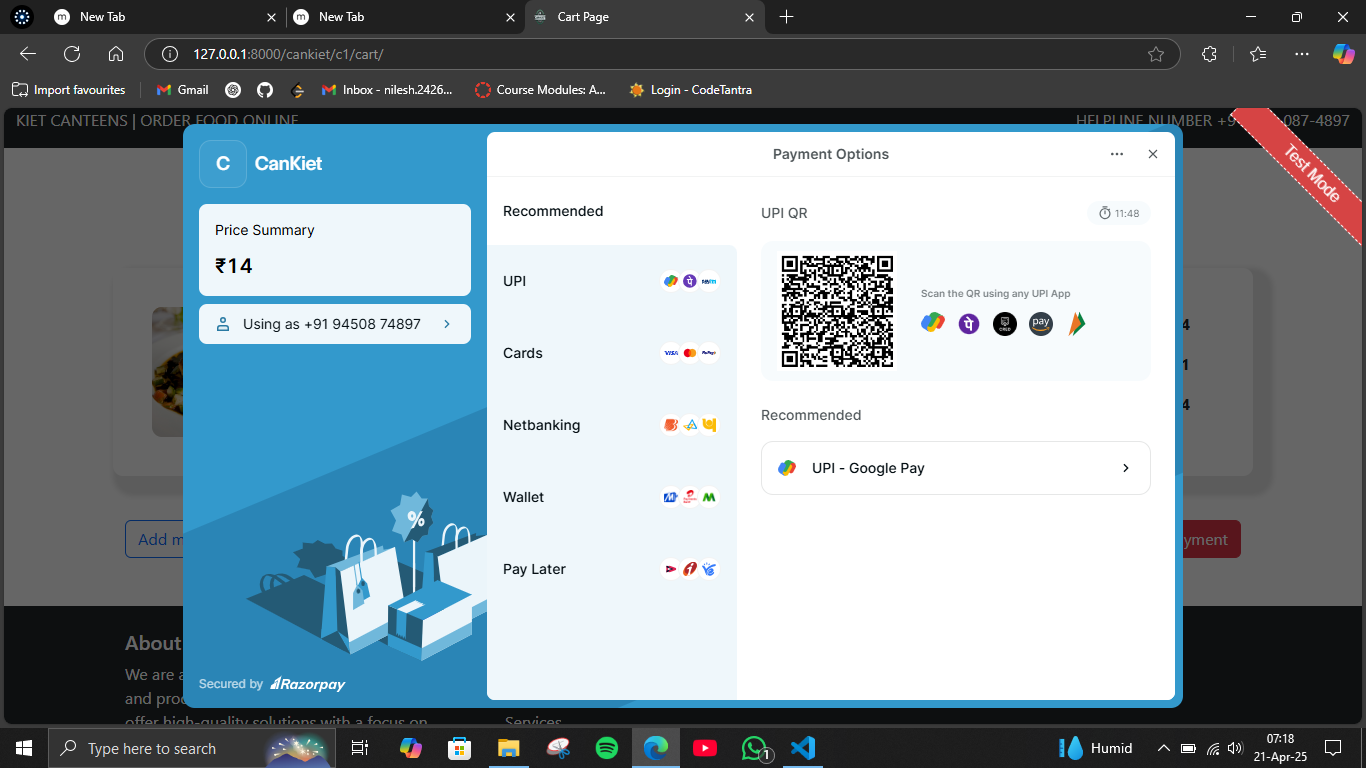


Fig 6.6 Payment page

Figure 6.6 shows the payment page of the CanKiet online canteen. It displays the total order amount and provides fields for the user to enter their card details, including the card number, expiry date, and CVV. Once the user has entered their payment information, they can click on the "Pay Now" button to proceed with the payment.

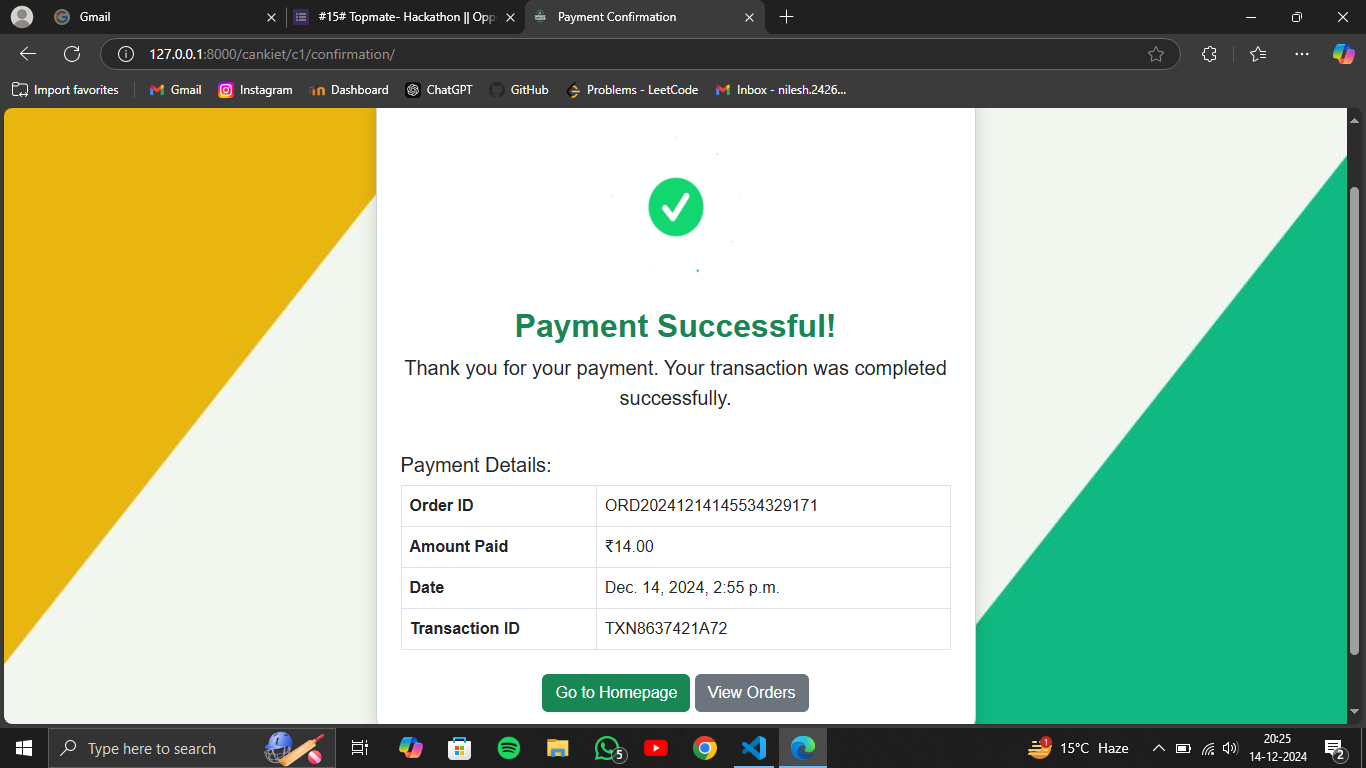


Fig 6.7 Confirmation Page

Figure 6.7 shows the payment confirmation page of the Can Kiet online canteen. It displays a message confirming the success of the payment, along with the order details, including the order ID, amount paid, date of transaction, and transaction ID. The page also provides two buttons: "Go to Homepage" and "View Orders".

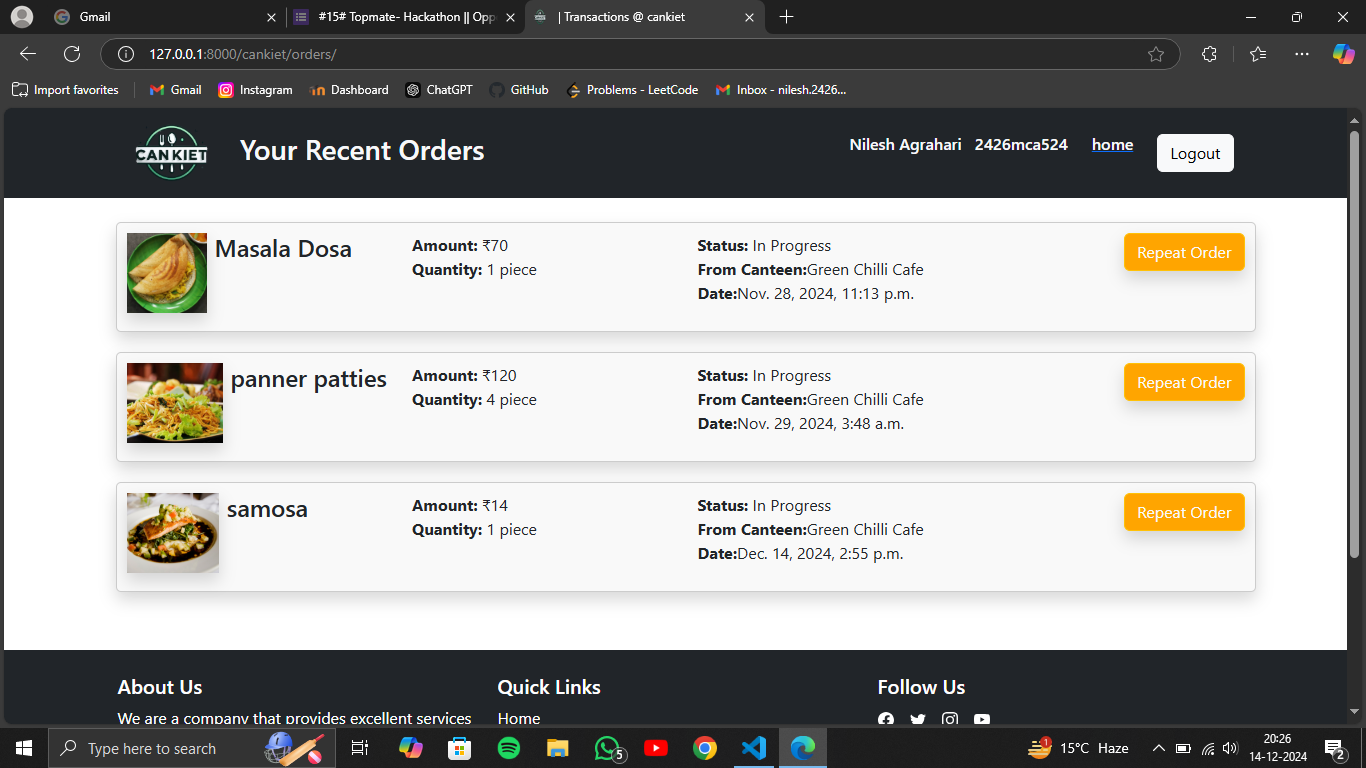


Fig 6.8 Orders Page

The webpage showcases a customer's recent food orders from CanKiet as shown in Figure 6.8, a food delivery service. The page displays a list of the customer's latest orders, each with details such as the item name, quantity, total amount, order status, restaurant name, and order date. Additionally, the page provides essential navigation links like "Home" and "Logout" for easy user interaction.

**Admin Dashboard**

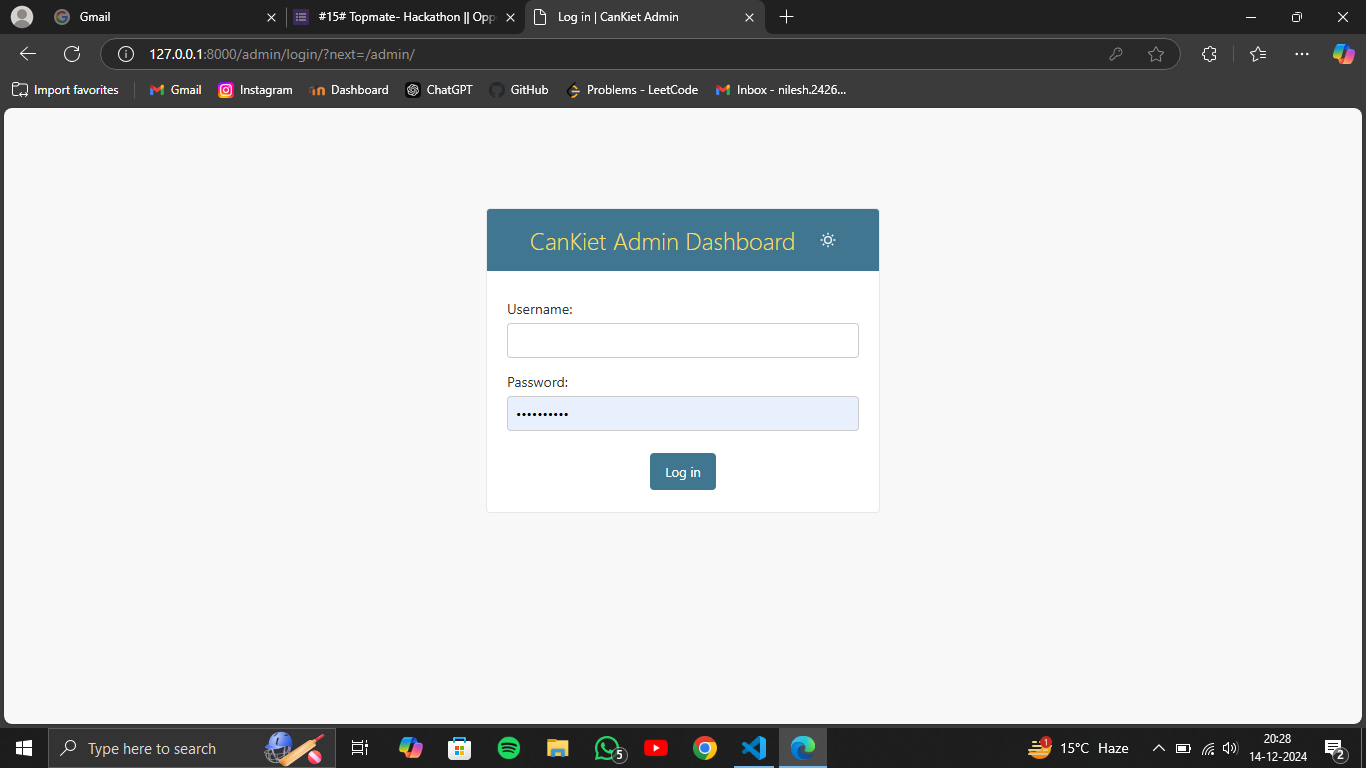


Fig 6.9 Admin login

The Figure 6.9 depicts a login page for the CanKiet Admin Dashboard. It features a straightforward design with a cantered form containing fields for username and password. Below the form, a "Log in" button is placed for authentication purposes. The page's overall aesthetic is clean and professional, with a focus on user experience and security.

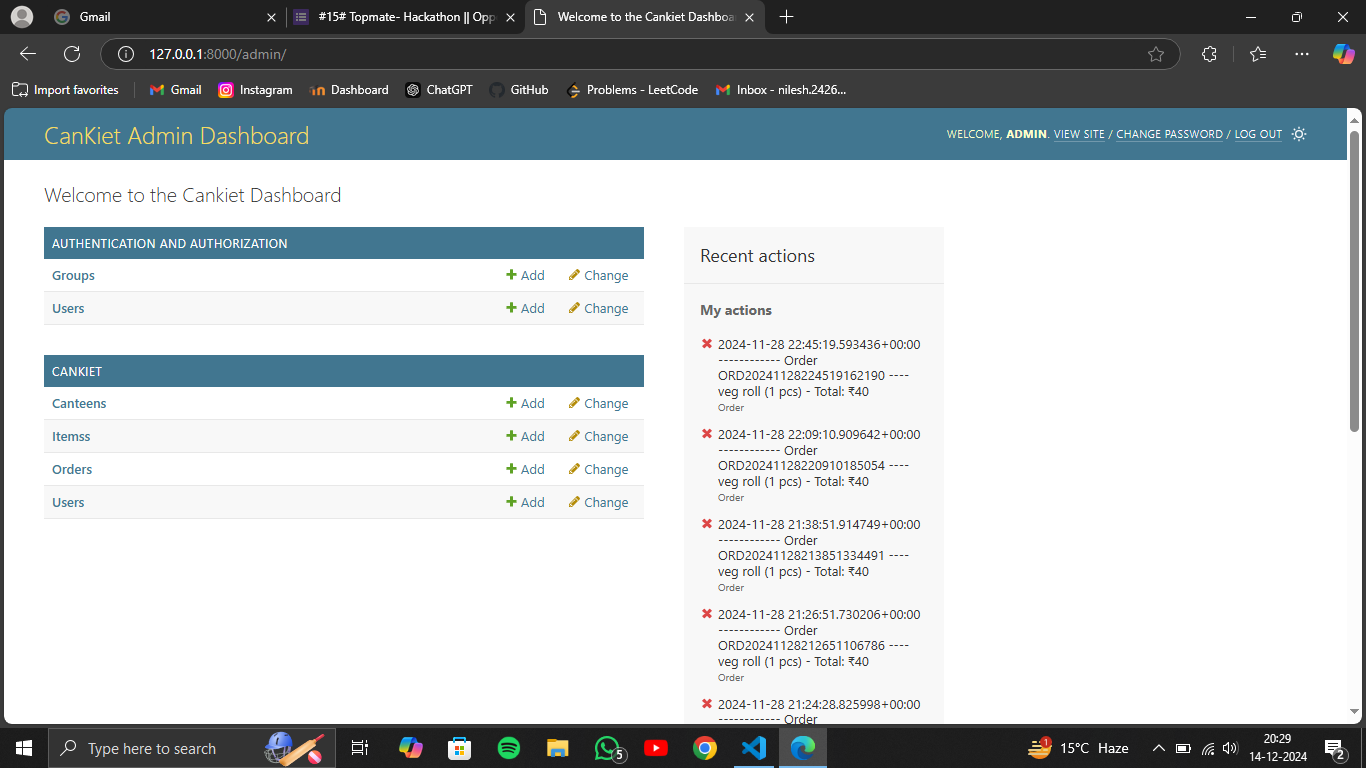


Fig 6.10 Admin Dashboard

The central section of the dashboard as shown in Figure 6.10 focuses on CanKiet-specific features, allowing the admin to manage canteens, items, orders, and users. Each section provides options for adding, changing, and viewing existing data. This comprehensive interface empowers the admin to effectively oversee and manage all aspects of the CanKiet food delivery service.

**REFERENCES**

1. [https://www.djangoproject.com](https://www.djangoproject.com/)
2. [https://getbootstrap.com](https://getbootstrap.com/)
3. Gupta, A., & Kumar, R. (2021). Smart Canteen Management Systems: A Comprehensive Review. International Journal of Computer Applications.
4. Kumar, S. (2020). Using Python and Django for Web Development. Tech Journal.
5. "Swiggy and Zomato Business Models," Business Insider, 2021.

**Bibliography**

1. Django Software Foundation. (n.d.). Django: The web framework for perfectionists with deadlines. Retrieved from <https://www.djangoproject.com>
2. Bootstrap. (n.d.). Bootstrap: The world’s most popular framework for building responsive, mobile-first sites. Retrieved from <https://getbootstrap.com>
3. Gupta, A., & Kumar, R. (2021). Smart Canteen Management Systems: A Comprehensive Review. *International Journal of Computer Applications*, 183(25), 10-15.
4. Kumar, S. (2020). Using Python and Django for Web Development. *Tech Journal*, 12(3), 45-50.
5. Business Insider. (2021). Swiggy and Zomato Business Models. Retrieved from <https://www.businessinsider.com>
6. Razorpay. (n.d.). Razorpay Documentation: UPI Payment Integration. Retrieved [from https://razorpay.com/docs/](file:///C:\Users\hp\AppData\Local\Microsoft\Windows\INetCache\IE\Z1Y7J9EM\from%20https:\razorpay.com\docs\)
7. Python Software Foundation. (n.d.). Python Official Documentation. Retrieved from <https://www.python.org/doc/>
8. Sharma, P. (2022). Secure Payment Gateways in Web Applications. *Journal of Web Technologies*, 8(1), 22-30.
9. W3C. (n.d.). HTML5 Specification. Retrieved from <https://www.w3.org/TR/html5/>
10. Smith, J. (2019). Web Development with Django: A Practical Guide. *Packt Publishing*.