CHUKS KITCHEN

Backend Documentation

---

1. System Overview

Chuks Kitchen is a digital food ordering platform built for Mr. Chukwudi Okorie to manage his food business online. The system enables customers to register, browse meals, place orders, and track progress, while giving the admin team full control over the menu, orders, and status updates.

The backend is built using Django REST Framework (DRF) with PostgreSQL as the database, JWT (JSON Web Tokens) for authentication, and email-based OTP for account verification.

How the System Works End-to-End

Stage 1 - **Registration**: A new user visits the platform and creates an account using their email address or phone number.

Stage 2 - **Verification**: The system sends a 6-digit OTP to their email. They must enter this code to verify their account before they can log in.

Stage 3 - **Authentication**: Once verified, the user logs in and receives a JWT access token. This token is attached to every future request so the system knows who is making the request and what role they have (customer or admin).

Stage 4 - **Ordering**: The customer browses the publicly available food menu, selects items, and places an order. The system validates each item, calculates the total, and stores the order with a default status of pending.

Stage 5 - **Fulfilment**: The admin receives the order, marks payment as paid, then moves the order through the status lifecycle: confirmed => preparing => out_for_delivery => completed.

2. FLOW EXPLANATION

User Registration & Verification Flow

This flow handles everything from a new user creating an account to them successfully verifying their identity and gaining access to the platform.

Step-by-Step

- The user sends POST /api/auths/signup/ with identifier (email or phone), password, and optional referral code.
- The system checks whether the identifier contains '@' to determine if it is an email or phone number. This matters because it is with the email users receive OTP
- If the email or phone already exists in the database, signup is rejected immediately with a clear error message.
- If everything is valid, a User record is created with is_verified = False, and a random 6-digit OTP is generated and stored alongside a timestamp.
- The OTP is emailed to the user. It expires after 10 minutes
- The user submits their OTP to POST /api/auths/verify/. The system checks: does the user exist, is the OTP correct, has it expired?
- If all checks pass, is_verified is set to True, and the OTP is cleared from the database.
- The user can now log in via POST /api/auths/login/ and receives an access token (valid 24 hours) and refresh token (valid 7 days).

➔ **Food Menu Management Flow**

The menu system has two distinct audiences, the admin who manages it, and customers who browse it. These are kept on completely separate endpoints.

Admin Flow

- ➢ Admin logs in and receives a token with role: admin.
- ➢ Admin creates a Category first (e.g. Rice Dishes, Drinks) via POST /api/foods/admin/categories/.

- ➢ Admin then creates FoodItem records via POST /api/foods/admin/items/, linking each item to a category using the category's ID.
- ➢ Admin can update any food item at any time, changing price, description, or flipping is_available on or off, via PATCH /api/foods/admin/items/<id>/.

Customer / Public Flow

- ➢ Anyone (including unauthenticated visitors) can call GET /api/foods/ to see available meals.
- ➢ The view filters to only return items where is_available = True, so sold-out or hidden items never appear.
- ➢ The response includes category_name alongside the category ID so the frontend can display grouped menus without making extra requests.

➔ Order Placement & Management Flow

This is the most complex flow in the system. It handles cart validation, total calculation, order creation, payment logic, and the full status lifecycle.

Customer Order Placement

- ➢ Customer (must be logged in with role: customer) sends a POST to /api/orders/ with a list of items: [{ food_item_id, quantity }] plus delivery address and optional note.
- ➢ The serializer validates every item in the list BEFORE creating anything. If any single item fails (does not exist or is unavailable), the entire order is rejected. This prevents partial orders being saved.
- ➢ Once all items pass validation, the total price is calculated by multiplying each item's current price by its quantity and summing the results.
- ➢ The Order record is created with status: pending and payment_status: unpaid.
- ➢ Each item is saved as an OrderItem linked to the order, with the price snapshotted at that moment.

Payment Flow

- ➢ Every order starts as payment_status: unpaid.

- ➢ Admin marks payment received via PATCH /api/orders/admin/<id>/payment-status/ with { payment_status: paid }.
- ➢ Admin cannot confirm an order until payment is marked as paid, the system blocks this with a clear error message.
- ➢ This is logic-only (no payment gateway) but provides the correct structural foundation for integrating Paystack or Flutterwave later.

---

3. EDGE CASE HANDLING

Edge cases are unusual or failure scenarios that must be handled gracefully so the system never crashes, leaks data, or leaves data in an inconsistent state.

Edge Case	**What Happens**	**Where It Is Handled**
Duplicate email on signup	System rejects with: Email already registered	SignupSerializer.validate_identifier()
OTP is expired (>10 min)	System rejects with: OTP expired. Please request a new one	VerifyOTPSerializer.validate()
Wrong OTP entered	System rejects with: Invalid OTP	VerifyOTPSerializer.validate()
Login before verification	System rejects with: Account not verified	CustomTokenSerializer.validate()
Food unavailable at order time	Order rejected with item name: currently unavailable	PlaceOrderSerializer.create()
Food item deleted after cart	Order rejected with: Food item ID does not exist	PlaceOrderSerializer.create()

Empty cart submitted	Order rejected with: You must include at least one item	PlaceOrderSerializer.validate_items()
Customer cancels confirmed order	Rejected: Only pending orders can be cancelled	CustomerCancelOrderView
Admin updates completed order	Rejected: Cannot update a completed order	AdminUpdateOrderStatusView
Admin confirms unpaid order	Rejected: Payment has not been completed	AdminUpdateOrderStatusView
Payment update on cancelled order	Rejected: Cannot update payment on a cancelled order	AdminUpdatePaymentStatusView
Customer hits admin endpoint	Returns 403 Forbidden	IsAdmin permission class

---

4. ASSUMPTIONS

The following assumptions were made during development due to missing product requirements or technical information that was not specified in the brief.

Referral codes are accepted and stored but not validated against an existing referral system. No referral reward system was specified. The code is saved on the User record for future use.

Phone number users do not receive OTP via SMS. No SMS gateway was specified. Only email OTP is implemented. Phone signup stores the number but skips OTP.

Payment is logic-only with no real payment gateway. The project said 'assume logic only'.

Food images are optional. Admin may not have images for all items. ImageField is nullable so items can be created without photos.

Order cancellation by customer is only allowed while status is pending. Once the kitchen starts preparing food, cancellation causes real-world waste. This mirrors how real restaurant apps work.

---

5. SCALABILITY THOUGHTS

The current system is designed for a small food business with a modest number of daily orders. As the platform grows, specific parts of the architecture will need to evolve.

Users	**Priority Actions**
0 - 100	Launch as-is. Monitor errors. Ensure email delivery works reliably.
100 - 500	Add Celery for async emails. Add Redis caching on the menu endpoint.
500 - 2,000	Add persistent Cart model. Integrate real payment gateway (Paystack).
2,000 - 10,000	Horizontal scaling, read replicas, WebSocket notifications, S3 media storage.
10,000+	Full microservices consideration, dedicated search service, advanced monitoring.