AI-Powered Food Order WhatsApp Chatbot Documentation

Source Code : GitHub

Describe : Video

Project Overview

This document provides a comprehensive overview of the **AI-Powered Food Order WhatsApp Chatbot** application, detailing the architecture, key components, integration points, and the technologies used. The system is designed to automate restaurant order taking via WhatsApp and provide a real-time admin dashboard.

Feature	Technologies Used	Description
Backend	Java 17+, Spring Boot, JPA, Hibernate	Provides REST APIs, handles business logic, and manages the database.
Frontend	React JS (Vite), Tailwind CSS	Interactive, real-time dashboard for restaurant staff.
Chatbot	WhatsApp Cloud API, Gemini API	Handles messaging and intelligent conversational responses.
Payment	Razorpay Payment Gateway	Automates online payment and status tracking via webhooks.
Real-Time	WebSocket, STOMP, SockJS	Enables instant order updates on the admin dashboard.
Database	Oracle / MySQL (Configured for Oracle)	Stores menu items, orders, and order details.

Motivation for Custom Build

When questioned on why a custom solution (Spring Boot + React) was chosen over low-code or automation tools (like n8n, Zapier, Twilio Studio, or Make.com), the primary reasons are:

- **End-to-End Custom Control**: Low-code tools are restricted by predefined integrations. A full-stack solution offers complete control over backend logic, database, performance optimization, and custom APIs.
- **Scalability and Maintainability** : Spring Boot provides a **robust architecture** for enterprise-level applications, supporting modular structure, version control, and efficient performance at large scale, unlike simple automation workflows.
- **Integration Flexibility** : Custom code allows seamless integration with **any third-party service** (e.g., Razorpay, Gemini API, WhatsApp Cloud API) via REST, overcoming the limitations of pre-built connectors.
- **Security and Data Ownership** : For sensitive enterprise data, building a custom system ensures full control over authentication, authorization, and data storage, enforcing proper Spring Security and database encryption practices.
- **Cost and Vendor Independence** : Eliminates dependency on subscription models that charge per workflow or action, leading to long-term cost benefits for the organization.

⚙️ Backend Architecture (Spring Boot)

The application uses a modular Spring Boot structure to manage business logic, database persistence, and external API calls.

1. Database Schema

The core persistence is managed via JPA Entities, mapping to the following tables

Table	Purpose
Menu_Items	Restaurant's product catalog
Orders	Full customer order record
Order_Item	Junction table for Order and MenuItem

Complete DB Structure is described below.

2. Core Services

A. WhatsAppService (Chatbot Logic)

This service manages the conversation state (userStates and userData/userSessions maps) and handles API communication with the WhatsApp Cloud API and Gemini AI.

- **`sendMessage(toPhone, messageText)`**: Constructs and sends an HTTP POST request to the **WhatsApp Cloud API endpoint** (`https://graph.facebook.com/v17.0/{phoneNumberId}/messages`) using **RestTemplate**.
- **`handleIncomingMessage(payload)`**: Parses the incoming WhatsApp webhook payload to extract the user's phone number and message. It then uses a `switch(state)` block to manage the conversational flow.
    - **State Management**: Uses an in-memory `userStates` map (INIT, ASK_NAME, TAKE_ORDER, ASK_PAYMENT) to track where each user is in the ordering process.
    - **Auto Session Expiry**: Implements a background thread to clear inactive user sessions after **10 minutes** to prevent memory buildup.
- **`getGeminiAIResponse(userMessage)`**: Sends the user's message to the Gemini API and parses the AI's response.

B. OrderService (Business Logic)

Handles the persistence and retrieval of order data.

- **`saveOrder(...)`**: Creates an `Order` entity and corresponding `OrderItem` entities, calculates the `totalPrice` from menu item prices and quantities, and saves the complete order structure to the database.
- **Repository Layer**: Uses **JPA Repositories** (`OrderRepository`, `MenuItemRepository`, `OrderItemRepository`) for database interaction.

C. RazorpayService (Payment Logic)

- **`createPaymentLink(...)`**: Uses the Razorpay Java SDK to generate a dynamic payment link for the customer's order. It ensures the amount is converted to **paisa** before sending the request and embeds an internal `ResturantOrder_ID` in the `notes` for later lookup.

3. API Communication Details

Component	Endpoint / Action	Tool	Purpose
WhatsApp Message Send	POST https://graph.facebook.com/.../messages	RestTemplate	Bot sends a message to the user[26].
WhatsApp Message Receive	@PostMapping("/webhook")	WhatsAppController	Endpoint for Meta to send incoming user messages to the bot (Webhook)
Gemini AI Call	POST https://generativelanguage.googleapis.com/.../generateContent	RestTemplate	Bot sends u + system co Gemini for a intelligent re
Payment Callback	@PostMapping("/api/payment/callback")	RazorpayController	Webhook endpoint for Razorpay to confirm payment status instantly.

🧠 Gemini AI Integration & Chat Logic

1. Gemini API Payload Structure

The application builds a specific JSON payload to communicate with the Gemini API to generate content:

```
{
  "contents": [
    {
      "parts": [
        {"text": "User message here"}
      ]
    }
  ],
  "generationConfig": {
    "temperature": 0.7,
    "maxOutputTokens": 1000
  }
}
```

Problem & Resolution: Initially, incorrect JSON structures (using old fields like `input` or `prompt.text`) caused `400 Bad Request` errors. The fix was adopting the required `contents` + `generationConfig` structure and using a mutable `HashMap` for construction.

2. AI Conversational Rules

To ensure the AI behaves as a specialized restaurant assistant (and not a general chatbot), strict system instructions are passed in the prompt:

You are a friendly and professional restaurant assistant for "The Craving" on WhatsApp.So, Behave accordingly.

Your role is to help customers with:

- Answering questions about the restaurant (working hours, menu, order status etc.)
- Providing information about ordering process
- Handling general food-related inquiries
- Being warm, welcoming, and conversational like a real restaurant staff member

CONVERSATION STYLE:

- Talk naturally like a helpful staff member, not a robot
- Keep responses SHORT (2-3 sentences max)
- Use emojis naturally but sparingly
- Answer questions directly without being repetitive
- Always try to keep the conversation crisp and small
- You are adding too many "*", just add that required , avoid unnecessary. Use bold only where necessary.

CRITICAL RULES:

1. ONLY mention items that are in the menu list below and give the menu category wise and in formatted text .
2. If customer asks about an item NOT in the menu, say "Sorry, we don't have [item] today. Type 'Order' to see what's available!"
3. If customer asks about items IN the menu, confirm briefly and tell them to type 'Order' to place an order
4. For order status questions: Tell them to type 'status [Order ID]' (e.g., 'status 123') to track their order
5. For off-topic questions (sports, politics, etc.): Polite redirect - "I'm here to help with food orders! 😊 Type 'Order' to get started."
6. For location/hours/contact: Say "For more details, please type 'Order' to start ordering!"
7. NEVER make up or assume menu items or order statuses - only use the info below
8. Be conversational - vary your responses
9. ALWAYS end by prompting the user to type 'Order' to start ordering and the text 'Order' should be in bold.
10. Always follow WhatsApp messaging policies
11. If the user ask to cancel the order during ordering, tell them to type 'Cancel'. If order is already confirmed, tell them to contact restaurant
12. NEVER provide false information about the restaurant

13. Always encourage the customer to type "Order" to start ordering
14. When customer places order, they will receive an Order ID which they can use to track status
15. Whenever you are mentioning the restaurant name , Make sure that the restaurant name should be bold for whatsapp formatting.

Restaurant Info:

- Name: The Craving
- Specialty: Delicious food, quick service
- Payment: Cash, UPI, Card
- Order Tracking: Type 'status [Order ID]' (e.g., 'status 123') to track your order

Now respond briefly and naturally to the customer's question based ONLY on the menu above.

Remember: Only show full menu if customer specifically asks for it!

💳 Razorpay Payment Integration

The application integrates with Razorpay to accept online payments securely.

1. Implementation Flow

1. **Link Generation**: When the user selects a non-Cash payment method, the backend calls **`RazorpayService.createPaymentLink()`**.
2. **Amount Conversion**: The total amount is converted from Rupees to **Paisa** (`amount * 100`) as required by Razorpay.
3. **Customer Link**: Razorpay returns a short payment URL (`https://rzp.io/i/abc123`), which the bot sends to the customer on WhatsApp.
4. **Webhook Setup**: A public callback URL (e.g., `https://your-ngrok-url.ngrok-free.dev/api/payment/callback`) is configured in the Razorpay Dashboard to receive real-time payment events.
5. **Status Update**: The `RazorpayController` handles the incoming webhook. It extracts the internal `ResturantOrder_ID` from the payload's `notes` and updates the `Order` status in the DB to **`CONFIRMED`** or **`PAYMENT_FAILED`**.
6. **Duplicate Fix**: Duplicate webhooks are ignored if the order status is already `CONFIRMED`.

In the Indian digital payments ecosystem, several payment gateways are widely used—such as Paytm, PayU, CCAvenue, Cashfree, and Instamojo. Each of these platforms provides essential features like UPI, cards, net banking, refunds, and merchant dashboards.
Despite this competition, **Razorpay has become the most preferred choice among developers, startups, and modern applications**.

For this project, Razorpay was selected as the primary payment gateway due to its superior developer experience, ease of integration, highly reliable system architecture, and strong ecosystem support.

2. Why Razorpay Was Selected

2.1 Developer-Friendly Integration

Razorpay is considered the most developer-friendly gateway in India.

A major advantage of Razorpay is its **simple and elegant API design (Easy integration SDK documentation)**.
Many gateways offer similar payment features, but their implementations are often:

- Complex
- Poorly documented
- Require manual hashing
- Lack standardization

Razorpay provides:

- Clean REST APIs
- Clear documentation
- Easy server-to-server and client-side integration
- Fast onboarding with an instant test environment

Because of this, the **integration time reduces drastically**, which is beneficial during development, testing, and project demonstrations.

✓ Easy integration with:

- Spring Boot
- Node.js
- React
- Android
- PHP
- Python

Example:

- Creating an order in Razorpay is just a single API call.
  Other gateways require multiple steps and complex auth flows.
- This is why developers prefer Razorpay for projects, demos, and production systems.

2.2 Comprehensive Payment Method Support

Razorpay supports **every major payment mode**, including:

- UPI
- Debit/Credit cards
- Net banking
- Wallets
- Pay Later/EMI
- International payments

While other gateways support many of these options, Razorpay ensures **better compatibility and smoother execution** across all platforms.

2.3 Superior Checkout Experience

Razorpay provides one of the best **pre-built checkout UIs** in India. The checkout interface is:

- Modern and responsive
- Optimized for mobile and web
- Fast and secure
- User-friendly with minimal redirects
- Minimal redirects
- Fast UPI flow
- Auto bank detection
- Error handling
- Works well on mobile + web

This enhances end-user satisfaction and **significantly increases payment success rates**, which is a critical factor in production systems.

2.4 Fast Merchant Onboarding & Easy Activation

Unlike many traditional gateways that require:

- Lengthy verification processes
- Multiple business documents
- Security deposits
- Long waiting periods

Razorpay offers:

- Instant test mode
- Quick activation for live mode
- Smooth KYC workflow

This makes Razorpay ideal for **startups, small businesses, prototypes, and engineering projects**.

2.5 Stable Webhooks and Reliable Events

For real-time applications—like order confirmation, refunds, or payment updates—webhook stability is essential.

Compared to Paytm, PayU, and some legacy gateways, Razorpay offers:

- Fast and consistent webhook delivery
- Detailed logs
- Secure signature verification
- Retry logic for failed deliveries

This reliability is crucial for **backend automation, order handling, and financial accuracy**.

2.6 Strong Security Standards

Razorpay is:

- PCI DSS Level 1 compliant
- Equipped with tokenization
- Protected with fraud detection models
- Enforced with strong encryption and compliance

These features reduce the security burden on developers and improve customer trust.

2.7 Complete Payment Ecosystem

Razorpay doesn't just offer payment gateway services—it provides a **full suite** of financial tech tools in one place:

- Payment Links
- Payment Pages
- Subscriptions & Recurring Billing
- Smart Collect (Virtual UPI IDs & VPA-based collections)
- Invoices
- Settlements Dashboard
- Refund APIs
- QR Code Payments

Other gateways might offer some of these features, but **Razorpay provides all of them in a unified, easy-to-use ecosystem**.

2.8 Strong Dashboard & Analytics

The Razorpay Merchant Dashboard provides:

- Real-time transaction insights
- Settlement history
- Payouts tracking
- Refund management
- Dispute resolution
- Easy test/live mode switching

This greatly helps during both development and production monitoring.

2.9 High Trust in the Developer and Startup Community

Razorpay is widely adopted by:

- Startups

- SaaS platforms
- E-commerce businesses
- D2C brands
- Educational platforms
- Finance products

This community trust ensures better resources, tutorials, support, and integrations—which ultimately helps developers build faster.

Even though other payment gateways also provide similar features, Razorpay stands out because of its clean API design, easy integration with modern stacks, superior checkout experience, fast onboarding, reliable webhook system, and complete ecosystem of payment solutions. These advantages make Razorpay the most developer-friendly and startup-friendly payment gateway in India. For these reasons, it was chosen for this project to ensure smooth development, high reliability, and better user experience.

❓ *"Why didn't you choose Paytm / PayU / CCAvenue / Cashfree instead of Razorpay?"*
or
❓ *"These providers also give all features, so what makes Razorpay better?"*

Yes, other providers like Paytm, PayU, CCAvenue and Cashfree also offer major features like UPI, cards, net banking and APIs. But in real development, the biggest difference is the developer experience and integration simplicity. Razorpay's APIs are significantly cleaner, better documented, easier to test, and faster to integrate compared to others.

Also, Razorpay provides a more modern checkout UI, higher UPI/card success rates, stable webhooks, faster onboarding, and a complete ecosystem (payment links, smart collect, subscriptions) in one place. Because of these practical advantages, Razorpay is the most developer-friendly and startup-friendly gateway in India. So even though others provide similar features, Razorpay gives the best overall experience for both developers and users.

✓ "Why not Paytm Gateway?"

Paytm is strong, but they prioritize their wallet and have stricter onboarding. Razorpay gives smoother UPI integrations and easier developer testing.

✓ "Why not PayU?"

PayU's API requires multiple steps and custom hash generation, making integration slower. Razorpay's order–payment–webhook flow is cleaner and faster.

✓ "Why not CCAvenue?"

CCAvenue is powerful but their APIs feel outdated, UI is less modern, and onboarding is lengthy compared to Razorpay.

✓ “Why not Cashfree?”

Cashfree is good for payouts, but Razorpay has better checkout UX, higher success rates, and a more complete payment ecosystem.

All gateways provide similar features, but Razorpay provides the best combination of clean APIs, smoother checkout, reliable webhooks and fastest integration — that’s why it’s the first choice for developers.

“But all are the same, Why this ?”

Technically they all support similar payment modes, but in software engineering, developer experience, integration simplicity, reliability and support matter more than just features. Razorpay performs better in all those practical areas

---

🖥️ Frontend (React) Dashboard

The restaurant administration panel is built with React JS to provide a real-time, interactive interface for managing orders.

1. Real-Time Order Updates (WebSocket + STOMP)

The dashboard achieves instant order updates using a publish-subscribe architecture:

Component	Protocol / Action	Purpose
Connection	new SockJS("http://localhost:8080/ws")	Establishes a persistent, cross-browser connection to the backend.
Protocol	stompClient.over(socket)	STOMP (Simple Text Oriented Messaging Protocol) adds structure and topic-based routing over the raw WebSocket channel.
Subscription	stompClient.subscribe("/topic/orders", ...)	The frontend client subscribes to the topic where the backend broadcasts new order events.
Backend Trigger	simpMessagingTemplate.convertAndSend("/topic/orders", order)	The Spring Boot backend instantly pushes the new order JSON to all subscribed dashboards.

2. Analytics and Reporting

The dashboard provides monthly summary and AI-driven insights:

Feature	Data Source / Technology	Details
Monthly Summary	REST API (/api/orders/monthly-summary) , Custom SQL Query	Aggregates and displays orders, revenue, average order value, and payment mode split (Cash/Online) per month, excluding cancelled orders.
AI Insights	REST API (/api/ai-insights), Gemini API	Backend passes analytics data to Gemini with a prompt to generate plain-English business insights (e.g., "This week's sales are 12% higher...").
Charting	**Recharts** Library	Used for visual representations like Daily Revenue Trend (Line Chart), Payment Split (Pie Chart), and Popular Items (Bar Chart).
CSV Export	react-csv Library	Allows one-click export of transaction history data into an Excel-friendly CSV file.

3. Menu Management

The frontend includes a management tab allowing administrators to perform CRUD operations on menu items:

Action	Path/Method	Purpose
View All	GET /api/menu	Fetches the complete menu list.
Add/Edit	POST/PUT /api/menu	Saves new items or updates existing ones.
Toggle Status	PATCH /api/menu/{id}/toggle	Instantly switches an item's availability status (Available / Out of Stock).

The **HTTP PATCH method** is typically used when you need to apply **partial modifications** to a resource, rather than replacing the entire resource with a PUT request.

In this specific case, PATCH is used because:

1. **Partial Update**: You are only changing **one field** (`available`) of the `MenuItem` entity. A `PUT` request would typically require sending the entire menu item object (name, description, price, etc.).
2. **Toggle Functionality**: The purpose is a one-click action—**Toggle availability switch**—which is a lean operation that doesn't need the full data payload.
3. **Efficiency and Clarity**: Using PATCH makes the API endpoint's purpose clear (it's a focused update) and is generally more efficient than a PUT for minor changes. The action performed is `toggleItem`.
4. **Instant Status Change**: It allows the restaurant owner to quickly change an item's status in one click, reflecting the change live in the UI. The action is simple and results in an immediate `Status updated` toast notification.

🔑 Setup Prerequisites

To run and deploy the chatbot, the following third-party credentials and setup steps are required:

1. **WhatsApp Cloud API**:

   - Obtain from "Meta For Developers"
   - URL : https://developers.facebook.com/apps/

   - Get a **Phone Number ID** and **Permanent Access Token** from the Meta Developers portal. Add ad verify Token by yourself in configuration page that will be used in backend to verify the request is authenticated and add the backend whatapp api webhook in the webhook section in the configuration page.
     - Verify your phone number and business account.
     - Configure the Webhook URL using a public-facing URL (e.g., from **ngrok**) since the WhatsApp API cannot access localhost directly.
2. **Razorpay**:

   URL : https://dashboard.razorpay.com/app/dashboard

   - Generate a **Key ID** and **Key Secret** from the Razorpay Dashboard.
   - Configure the Webhook URL in Razorpay Dashboard to your public URL (e.g., /api/payment/callback) and subscribe to the necessary payment events.
3. **Google Gemini AI**:
   - Obtain a **Google API Key** from your Google Cloud Project to access the Gemini API.
   - URL : https://aistudio.google.com/api-keys

Q. **"There are automation tools like n8n, Zapier, or Twilio Studio that can create such bots easily. Why did you choose to build yours from scratch using Java Spring Boot and React?"**

Q. "Couldn't you have implemented this using existing low-code or automation tools? Why go with a full-stack custom solution?"

Q. “What advantage does your Spring Boot + React implementation offer over using ready-made automation platforms?”

Q. “If your goal was automation, why didn’t you use platforms like n8n or Make.com instead of writing so much code?”

Q. “What was the main motivation behind using Java and React when existing automation tools can already do similar things?”

ANSWER

◇ 1. End-to-End Custom Control

“Tools like n8n or Zapier are great for quick prototyping, but they are limited by predefined nodes and integrations.
With Spring Boot and React, I have full control over every component — backend logic, APIs, database, and UI.
That means I can customize workflows, optimize performance, and extend functionality exactly as needed.”

💡 *In short:* “Automation tools are low-code; I wanted a full-code, customizable system.”

◇ 2. Scalability and Maintainability

“Enterprise applications require scalability, version control, and maintainability.
Spring Boot provides a robust backend architecture, REST APIs, and modular structure.
React offers reusable UI components and fast rendering — which automation tools can’t handle efficiently at large scale.”

💡 *You show that your design is ready for **real-world production**, not just demos.*

◇ 3. Integration Flexibility

“Using Spring Boot, I can integrate any third-party service — Razorpay, Gemini API, or even WhatsApp Cloud API — through REST.
Automation tools are restricted to whatever connectors they provide.
But our approach allows seamless integration with any API or microservice.”

💡 *This shows you understand real integration challenges.*

◇ 4. Learning & Skill Demonstration

"As a developer, building from scratch with Spring Boot and React helps me demonstrate deep understanding of backend, frontend, and API communication.
It's not just about creating a workflow — it's about understanding how everything works under the hood."

💡 *Perfect justification for a fresher or project portfolio.*

◇ 5. Security and Data Ownership

"In enterprise applications, we can't rely on third-party tools for sensitive data.
Building our own system gives full control over authentication, authorization, and data storage — following proper Spring Security and database encryption practices."

💡 *This is a major reason real companies prefer custom-built solutions.*

◇ 6. Cost and Vendor Independence

"Automation platforms usually charge per workflow or per action.
A custom Spring Boot system eliminates that dependency and provides long-term cost benefits for organizations."

☑ Short, Interview-Ready Answer (2 lines):

"Tools like n8n are good for basic workflows, but they're limited and not scalable.
I built it with Java Spring Boot and React to gain full control, scalability, API flexibility, and to strengthen my backend-frontend development skills."

NOTE :

YOU NEED TO HAVE YOUR PERMANENT ACCESS TOKEN OF YOUR WHATSAPP API FROM BELOW

URL : https://developers.facebook.com/apps/

YOU NEED TO HAVE CREATE A BUSINESS PORTFOLIO IN THE FACEBOOK BUSINESS

ALSO NEED TO ADD AND VERIFY YOUR PHONE NO. WHICH YOU WANT TO USE AS BOT AND VERIFY YOUR BUSINESS

While in development phase our Springboot App will run in local server i.e, **localhost** , As WhatsApp API can't access localhost webhook, we need to make our localhost url public. Here, I am using **ngrok** to make the url public. It will give you a redirected url which will work

as public url for your localhost url. Every request to that public url will be redirected to your localhost.

PROJECT START :

DB Structure

1. Menu_Items Table

Column	Type	Constraints	Description
id	NUMBER	PRIMARY KEY	Unique menu item ID
name	VARCHAR2(100)	NOT NULL	Name of the menu item
description	VARCHAR2(255)		Short description of the item
price	NUMBER(10,2)	NOT NULL	Price of the item
available	NUMBER(1)	DEFAULT 1	Availability (1 = Yes, 0 = No)

2. Orders Table

Column	Type	Constraints	Description
id	NUMBER(19)	PRIMARY KEY, NOT NULL	Unique order ID
customer_name	VARCHAR2(255)	NOT NULL	Customer's name
user_phone	VARCHAR2(255)	NOT NULL	Customer's phone number
order_time	TIMESTAMP(6)		Time when the order was placed
payment_mode	VARCHAR2(255)		Payment mode (Cash, UPI, Card, etc.)
status	VARCHAR2(255)		Order status (e.g., PENDING, CONFIRMED, DONE)
total_price	FLOAT(53)		Total price of the order
order_status	VARCHAR2(255)		Additional status or delivery stage (if any)

3. order_ItemTable

Column	Type	Constraints	Description
id	NUMBER	PRIMARY KEY	Unique ID for order item
order_id	NUMBER	FOREIGN KEY REFERENCES Orders(id)	Linked order ID

menu_item_id	NUMBER	FOREIGN KEY REFERENCES MenuItem(id)	Linked menu item ID
quantity	NUMBER	NOT NULL	Quantity of the menu item in order

Application.properties

spring.application.name=AI-powered-WhatsApp-ChatBot

#DataSource confg

spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver

spring.datasource.url=jdbc:oracle:thin:@MSI:1521:orcl

spring.datasource.username=system

spring.datasource.password=Bhagyajyoti768

#JPA-Hibernate Properties

spring.jpa.database-platform=org.hibernate.dialect.OracleDialect

spring.jpa.show-sql=true

spring.jpa.hibernate.ddl-auto=update

WhatsApp Cloud API

whatsapp.phoneNumberId=YOUR_PHONENUMBERID //Get from meta business portal for you account

whatsapp.accessToken=YOUR_WHATSAPP_ACCESS_TOKEN //Get from meta business portal for you account

In "WhatsAppService" class

package com.chatBot.service;

import java.util.Map;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.http.HttpEntity;

import org.springframework.http.HttpHeaders;

```
import org.springframework.http.MediaType;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

@Service
public class WhatsAppService {
        @Value("${whatsapp.phoneNumberId}")
        private String phoneNumberId;

        @Value("${whatsapp.accessToken}")
        private String accessToken;
        private RestTemplate restTemplate= new RestTemplate();
        /*
         * RestTemplate is a Spring class used to send HTTP requests and receive HTTP responses from another server.
         *It lets your Spring Boot application act like a client calling APIs.
         */
        public void sendMessage(String toPhone, String messageText)
        {
                String url="https://graph.face        book.com/v17.0/" + phoneNumberId + "/messages";
                HttpHeaders headers= new HttpHeaders();
                headers.setBearerAuth(accessToken);
                headers.setContentType(MediaType.APPLICATION_JSON);
                Map<String, Object> payload = Map.of(
                "messaging_product", "whatsapp",
                "to", toPhone,
                "type", "text",
                "text", Map.of("body", messageText)
            );
```

```
                HttpEntity<Map<String, Object>> request = new HttpEntity<>(payload,
headers);
            restTemplate.postForEntity(url, request, String.class);
        }
        public void handleIncomingMessage(Map<String, Object> payload) {
            // TODO: Parse WhatsApp message and handle conversation state
        }
}
```

https://graph.facebook.com/v17.0/" + phoneNumberId + "/messages

1. What is this URL?

This is the WhatsApp Cloud API endpoint provided by Meta (Facebook).

It's used to send messages from your WhatsApp Business account (or sandbox test number) to a user.

The structure is:

https://graph.facebook.com/<API_VERSION>/<PHONE_NUMBER_ID>/messages

Where:

<API_VERSION> → version of the Graph API you're using, e.g., v17.0

<PHONE_NUMBER_ID> → your WhatsApp bot number's ID (the "From" number)

/messages → endpoint to send a message

private final RestTemplate restTemplate = new RestTemplate();

1. What is RestTemplate?

RestTemplate is a Spring class used to send HTTP requests and receive HTTP responses from another server.

It lets your Spring Boot application act like a client calling APIs.

2. Why we use it here

In our bot, we need to send messages to WhatsApp Cloud API.

WhatsApp Cloud API expects an HTTP POST request with JSON payload.

RestTemplate is used to:

- Build the request
- Set headers (like authorization token)
- Send it to the WhatsApp API
- Receive the response

1. HttpHeaders headers = new HttpHeaders();

- Creates a **container for HTTP headers**.
- HTTP headers carry metadata for the request, like authorization and content type.

---

2. headers.setBearerAuth(accessToken);

- Sets **Authorization header** using **Bearer token**.
- WhatsApp Cloud API requires an **access token** to verify that your bot is allowed to send messages.
- This translates to HTTP header:
- Authorization: Bearer <accessToken>

---

3. headers.setContentType(MediaType.APPLICATION_JSON);

- Tells the API that **we are sending JSON data** in the request body.
- WhatsApp Cloud API expects message data as JSON.

---

4. Creating the Payload

```
Map<String, Object> payload = Map.of(
   "messaging_product", "whatsapp",
   "to", toPhone,
   "type", "text",
   "text", Map.of("body", messageText)
);
```

- This builds the **actual message body** we send to WhatsApp.

- Fields explained:
    - "messaging_product": "whatsapp" → API knows this is a WhatsApp message.
    - "to": toPhone → Recipient phone number (user who sent the message).
    - "type": "text" → Type of message (text, image, etc.).
    - "text": {"body": messageText} → The actual text content of the message.

---

5. Wrapping Payload + Headers

HttpEntity<Map<String, Object>> request = new HttpEntity<>(payload, headers);

- HttpEntity bundles:
    1. **Body** → the JSON payload
    2. **Headers** → Authorization & Content-Type
- This is what RestTemplate sends as a full HTTP request.

---

6. Sending the Request

restTemplate.postForEntity(url, request, String.class);

- postForEntity() → Sends a **POST request** to the given URL.
- url → WhatsApp Cloud API endpoint (https://graph.facebook.com/<PhoneNumberID>/messages)
- request → Contains JSON payload + headers
- String.class → We expect a **response as a string** from the API (optional; mainly for logging or debugging).

✅ **Effect:**
This line sends a text message from your bot to the user via WhatsApp Cloud API.

In "WhatsAppController" class

package com.chatBot.controller;

import java.util.Map;

import org.springframework.http.HttpStatus;

```
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import com.chatBot.service.WhatsAppService;

@RestController
public class WhatsAppController {
	private final WhatsAppService whatsAppService;
private static final String VERIFY_TOKEN = "AI-chatBot-secret-token-07";
	//Initializing the fènal field via constructor
	public WhatsAppController(WhatsAppService whatsAppService)
	{
		this.whatsAppService=whatsAppService;
	}
	 // Webhook for receiving messages
	/*
	 * A webhook is a way for an app to provide other applications with real-time information.
	 * In this case, WhatsApp will send an HTTP POST request to this endpoint whenever a new message is received by your WhatsApp Business number.
	 * This allows your application to react immediately to incoming messages.
	 */
	@PostMapping("/webhook")
	public ResponseEntity<String> receiveMessage(@RequestBody Map<String, Object> payload)
	{
		whatsAppService.handleIncomingMessage(payload);
		return ResponseEntity.ok("EVENT_RECEIVED");
```

```
        }


        // Verification endpoint for WhatsApp Cloud API
    @GetMapping("/webhook")
    public ResponseEntity<String> verifyWebhook(@RequestParam("hub.mode") String
mode,

        @RequestParam("hub.verify_token") String token,

        @RequestParam("hub.challenge")String challenge)
    {
        if("YOUR_VERIFY_TOKEN".equals(token))
        {
            return ResponseEntity.ok(challenge);
        }
        else {
            return ResponseEntity.status(HttpStatus.FORBIDDEN).body("Verification
failed");
        }
    }
}
```

🔑 **Steps to set up Verify Token**

1. **In your Spring Boot code**
   Define a string, e.g.
2. private static final String VERIFY_TOKEN = "my-secret-token-123";

Then check it in your controller:

```
if (VERIFY_TOKEN.equals(token)) {
    return ResponseEntity.ok(challenge);
}
```

3. **In Meta Developer Console** (https://developers.facebook.com → Your App → WhatsApp → Configuration → Webhook)
    - When you add your webhook URL (e.g., https://yourdomain.com/webhook)
    - It will ask for **Verify Token**
    - Enter the **same string** you used in your code (my-secret-token-123)
4. **Meta will test it**
    - Meta sends a GET request to your webhook with:
    - hub.mode=subscribe
    - hub.verify_token=my-secret-token-123
    - hub.challenge=123456789
    - Your code checks the token → if it matches, you return hub.challenge.
    - ✅ Verification passes.

🔎 **Why do we use Verify Token?**

1. **To prove ownership of your webhook endpoint**
    - Meta (WhatsApp Cloud API) doesn't know who owns https://yourdomain.com/webhook.
    - Anyone could claim that URL.
    - So Meta sends a **verification request** with the hub.verify_token you gave in console.
    - If your backend responds with the same token → ✅ proves that you own and control that endpoint.
2. **To prevent fake or malicious verification attempts**
    - Without verify token, any attacker could call your webhook and pretend to be Meta.
    - The token ensures **only Meta's verification** can succeed.
3. **One-time setup, then forgotten**
    - This process happens **only once when you register webhook**.
    - After that, Meta knows your endpoint is valid → and will keep sending real user messages.

🔹 **What are these and why are we using them?**

1. @RestController

- **What it is:** A Spring Boot annotation that marks this class as a REST API controller.
- **Why we use it:** So that this class can handle HTTP requests (GET, POST, etc.) and return responses in JSON/text format.

---

2. private final WhatsAppService whatsAppService;

- **What it is:** Dependency injection of a **service class** (your business logic).
- **Why we use it:** The controller shouldn't contain business logic. Instead, it delegates tasks to the WhatsAppService.

---

3. Constructor Injection

```
public WhatsAppController(WhatsAppService whatsAppService) {
   this.whatsAppService = whatsAppService;
}
```

- **What it is:** Spring injects the WhatsAppService bean automatically when the controller is created.
- **Why we use it:** This makes the controller **testable, modular, and clean**.

---

4. @PostMapping("/webhook") → receiveMessage()

- **What it is:** This is an endpoint that listens for incoming **messages/events from WhatsApp**.
- **Why we use it:** Whenever a user sends a message to your WhatsApp Business number, Meta sends a **webhook (HTTP POST request)** with the message data.
- **Functionality:**
    - Reads the message payload (@RequestBody Map<String, Object> payload).
    - Calls whatsAppService.handleIncomingMessage(payload) to process it.
    - Returns EVENT_RECEIVED to confirm to Meta that you received the event.

---

5. @GetMapping("/webhook") → verifyWebhook()

- **What it is:** This endpoint is used **only once during setup** of your webhook in the Meta Developer Console.

- **Why we use it:** Meta needs to verify that **you own this server/URL**. It sends a challenge request.
- **Functionality:**
    - Meta sends: hub.mode, hub.verify_token, hub.challenge.
    - If your token matches ("YOUR_VERIFY_TOKEN"), you return the challenge.
    - If it doesn't match, return 403 Forbidden.

🔹 **Why do we need this whole controller?**

Because without it, your server can't:

1. **Receive WhatsApp messages** sent by users.
2. **Verify with Meta** that your webhook is valid.
3. **Pass data to service layer** for business logic (e.g., auto-reply, storing in DB, processing orders).

✅ In short:

- @PostMapping("/webhook") → handles **incoming messages**.
- @GetMapping("/webhook") → handles **verification with WhatsApp Cloud API**.
- WhatsAppService → your **business logic** (what you do with the messages).

✅ **Next Step: Implement the WhatsAppService (business logic)**

This is where your bot actually **handles messages, extracts user info, and replies**.

We will do the following:

Step 1: Extract incoming message info

From the JSON payload sent by WhatsApp, we need to extract:

- **User phone number** → from field → treat this as customer phone number
- **Message text** → text.body field → what user typed

Step 2: Send message back to user

Step 3: Plan conversation flow

Your bot should:

1. Detect **first message** → ask for **name**
2. Ask **order** (menu)
3. Ask **payment method**
4. Confirm **order**

We will later add **state management per user** so it knows what step each customer is in.

WhatsApp + Gemini AI Integration

Objective:
We wanted to integrate Google's Gemini AI with our WhatsApp chatbot so that the bot can intelligently respond to user messages using AI-generated content.

Procedure to Integrate Gemini AI with WhatsApp Chatbot

1. **Setup WhatsApp Cloud API**
    - Create a WhatsApp Business account or use a sandbox number.
    - Get the **Phone Number ID** and **Access Token** from Meta (Facebook) developers portal.
    - Test sending basic messages via WhatsApp Cloud API to ensure connectivity.
2. **Receive Incoming Messages**
    - Create a Spring Boot controller to receive webhook events from WhatsApp.
    - Parse the incoming JSON payload to extract:
        - User phone number (from)
        - User message (text.body)
3. **Call Gemini AI API**
    - Get your **Google API key** from your Google Cloud project.

      URL : https://aistudio.google.com/api-keys
    - Use the Gemini API endpoint:
    - https://generativelanguage.googleapis.com/v1beta/models/gemini-2.5-flash:generateContent?key=<API_KEY>

- Prepare **JSON payload** in the required format:

```
{
 "contents": [
  {
   "parts": [
    {"text": "User message here"}
   ]
  }
 ],
 "generationConfig": {
  "temperature": 0.7,
  "maxOutputTokens": 200
 }
}
```

Send the POST request using RestTemplate or HttpClient.

4. **Parse Gemini Response**
    - Extract AI-generated text from the response path:
    - candidates -> content -> parts -> text
    - Handle cases where the response might be empty or invalid.
5. **Send AI Response to WhatsApp**
    - Use WhatsApp Cloud API /messages endpoint.
    - Send the AI-generated text back to the user.
6. **Manage Chat State (Restaurant-specific)**

- Keep track of conversation state per user (e.g., INIT, ASK_NAME, TAKE_ORDER, ASK_PAYMENT).
- **If user sends a message containing "order"**, the bot switches to restaurant mode:
    1. Ask for the user's name.
    2. Show the menu and ask for order choice.
    3. Ask for payment method.
    4. Confirm the order and send a summary.

- Store user data in memory (userData) for temporary tracking.

Problem Faced:

- Initially, our JSON payload format was **incorrect**.
    - Fields like input, max_output_tokens, or prompt.text were **not recognized** by the Gemini API.
    - This caused 400 Bad Request errors.
- The API responses either returned errors or an **empty response**, so the bot could not reply to the user.
- The mistake was mainly due to using the **old Gemini v1/v2 JSON structure** while the API expects the **new contents + generationConfig format**.

How We Resolved It:

1. Switched to the latest **correct Gemini API endpoint**:
2. https://generativelanguage.googleapis.com/v1beta/models/gemini-2.5-flash:generateContent
3. Built the **correct JSON payload**:

```
{
 "contents": [
  {
   "parts": [
    {"text": "User message here"}
   ]
  }
 ],
 "generationConfig": {
  "temperature": 0.7,
  "maxOutputTokens": 200
 }
}
```

4. Used a **mutable HashMap** to construct the payload so we could add generationConfig properly.
5. Updated response parsing to correctly extract text from candidates -> content -> parts -> text.
6. • Ensured the `RestTemplate` request uses `HttpEntity<Map<String, Object>>` with proper headers.
7. • For restaurant-specific flows, implemented chat states (`INIT`, `ASK_NAME`, `TAKE_ORDER`, `ASK_PAYMENT`) to handle orders separately from general AI chat.

Result:

- Now, incoming WhatsApp messages trigger a call to Gemini AI.
- AI generates a response correctly.
- WhatsApp bot sends the AI-generated response back to the user.
- Integration is **fully functional**.

Problem Faced with Gemini AI Payload

- **Code Snippet:**

```
Map<String, Object> textPart = Map.of("text", userMessage);

Map<String, Object> content = Map.of("parts", Collections.singletonList(textPart));


Map<String, Object> requestBody = new HashMap<>();

requestBody.put("contents", Collections.singletonList(content));

requestBody.put("generationConfig", Map.of(

    "temperature", 0.7,

    "maxOutputTokens", 1000

));
```

Issues Observed:

1. **Empty AI Response:**
    - Even though the request returned HTTP 200 OK, the candidates array in the response did not contain a text field.

- The bot was returning "🤖 AI returned an empty or invalid response." in the chat.

2. **Incorrect Parsing of Response:**
   - The response from Gemini API sometimes contains content with keys like role=model but no parts array if the request format is slightly off.
   - Using immutable maps (Map.of) for nested structures could cause problems when adding optional generation parameters later.
3. **Payload Compatibility Issue:**
   - The previous attempts with fields like "input" or "max_output_tokens" were rejected (400 Bad Request) because the Gemini API expects:
   - "*contents": [{ "parts": [{ "text": "..." }] }],*
   - "*generationConfig": { "temperature": ..., "maxOutputTokens": ...* }

Resolution:

- **Use Mutable Map:** new HashMap<>() for requestBody to safely add generationConfig.
- **Ensure Correct Nested Structure:** "contents" → "parts" → "text".
- **Correct Response Parsing:** Navigate through candidates -> content -> parts -> text to extract the AI-generated message.
- **Optional:** Increase maxOutputTokens to allow longer responses (e.g., 1000 instead of 200).

Outcome:

- The AI now reliably returns text in the chatbot.
- Longer responses are possible without truncation.

Creating JSON Payload for Gemini AI

1. **Step 1: Wrap User Message**
   - The user's WhatsApp message is wrapped in a parts array.
   - Each part is a map containing the key "text" with the user's message as value.

   Map<String, Object> textPart = Map.of("text", userMessage);

   Map<String, Object> content = Map.of("parts", Collections.singletonList(textPart));

2. **Step 2: Wrap Contents**

- The parts array is wrapped inside a contents array, as required by the Gemini API.

```
Map<String, Object> requestBody = new HashMap<>();
requestBody.put("contents", Collections.singletonList(content));
```

3. **Step 3: Add Generation Configuration**
   - Optional parameters like temperature and maximum output tokens are added under generationConfig.
   - **Temperature** controls randomness; **maxOutputTokens** controls response length.

```
Map<String, Object> generationConfig = Map.of(
  "temperature", 0.7,
  "maxOutputTokens", 1000
);
requestBody.put("generationConfig", generationConfig);
```

4. **Final JSON Structure:**
   The final JSON sent to the Gemini API looks like this:

```
{
 "contents": [
  {
   "parts": [
    { "text": "What should I eat for dinner?" }
   ]
  }
 ],
 "generationConfig": {
  "temperature": 0.7,
  "maxOutputTokens": 1000
 }
}
```

5. **Step 4: Sending the Request**
   - This map (requestBody) is wrapped in an HttpEntity with Content-Type: application/json headers and sent via RestTemplate POST request.

HttpEntity in Spring Boot is basically a **wrapper** for an HTTP request or response. It lets you bundle **headers** and **body** together so that RestTemplate (or other Spring HTTP clients) can send it properly.

Here's what it does:

1. **Holds the Body**
    - This is the actual content you want to send.
    - In our case, it's the JSON payload (requestBody) that contains the user message and configuration for Gemini AI.
2. **Holds HTTP Headers**
    - Metadata about the request, e.g., Content-Type: application/json, or Authorization tokens.
3. **Used in HTTP Requests**
    - You pass an HttpEntity to RestTemplate.postForEntity() or exchange() so Spring can send both the body and headers in a single HTTP request.

Example code:

HttpEntity<Map<String, Object>> request = new HttpEntity<>(requestBody, headers);

ResponseEntity<Map> response = restTemplate.postForEntity(url, request, Map.class);

- requestBody → The JSON we want to send.
- headers → Tells Gemini API that the body is JSON.
- request → Combines both, ready to be sent.

So, **without HttpEntity**, Spring would have no way of sending both headers and body together in one request.

```
private String getGeminiAIResponse(String userMessage) {

		try {
			String apikey = googleApiConfig.getApiKey();
			System.out.println("---Connecting to Gemini AI...");

			// Correct Gemini endpoint
			String url =
"https://generativelanguage.googleapis.com/v1beta/models/gemini-2.5-
flash:generateContent?key=" + apikey;

			HttpHeaders headers = new HttpHeaders();
			headers.setContentType(MediaType.APPLICATION_JSON);
```

```
                        // Request body
                        Map<String, Object> textPart = Map.of("text",
userMessage);
                        Map<String, Object> content = Map.of("parts",
Collections.singletonList(textPart));

                        Map<String, Object> requestBody = new HashMap<>();
                        requestBody.put("contents",
Collections.singletonList(content));
                        requestBody.put("generationConfig", Map.of(
                            "temperature", 0.7,
                            "maxOutputTokens", 1000
                        ));

                        HttpEntity<Map<String, Object>> request = new
HttpEntity<>(requestBody, headers);

                        //  ◇  Debug: print request body
                        System.out.println(">>> Gemini Request JSON: " + new
ObjectMapper().writeValueAsString(requestBody));

                        ResponseEntity<Map> response =
restTemplate.postForEntity(url, request, Map.class);

                        //  ◇  Debug: print raw response
                        System.out.println(">>> Gemini Raw Response: " +
response);

                        Map<String, Object> body = response.getBody();
                        if (body != null && body.containsKey("candidates")) {
                          List<Map<String, Object>> candidates =
(List<Map<String, Object>>) body.get("candidates");
                            if (!candidates.isEmpty()) {
                              Map<String, Object> firstCandidate = candidates.get(0);
                              Map<String, Object> contentMap = (Map<String,
Object>) firstCandidate.get("content");

                              if (contentMap != null) {
                                List<Map<String, Object>> parts = (List<Map<String,
Object>>) contentMap.get("parts");

                                if (parts != null && !parts.isEmpty() &&
parts.get(0).containsKey("text")) {
                                  return (String) parts.get(0).get("text");
                                } else {
                                  return "🤖 AI responded but didn't return any text
(finishReason="
                                      + firstCandidate.get("finishReason") + ")";
                                }
```

```
                        }
                    }
                }


            } catch (Exception e) {
                e.printStackTrace();
                return "🤖 AI is unavailable right now. Error: " +
e.getMessage();
            }

            return "🤖 AI returned an empty or invalid response.";
        }
```

When you set Content-Type to application/json and pass a Map as the body of an HttpEntity, Spring's RestTemplate automatically **converts the Map to a JSON string** before sending it.

- Map<String, Object> → becomes JSON like this:

Map<String, Object> map = Map.of("key", "value");

sends as:

{

"key": "value"

}

- This happens because Spring uses **Jackson** (the default JSON library) under the hood.
- You don't need to manually serialize it with ObjectMapper; RestTemplate handles it.

So yes, your requestBody Map is automatically converted to valid JSON in the HTTP POST request.

ResponseEntity<Map> response = restTemplate.postForEntity(url, request, Map.class);

we r calling or sending the request and getting response here

does the main work of sending and receiving data from Gemini AI:

restTemplate.postForEntity(...)

- Sends an HTTP POST request to the URL (url) with the given request body (request) and headers.
- here, the request body contains the JSON we built (the contents and generationConfig).

ResponseEntity<Map>

Captures the full HTTP response.

The Map.class tells Spring to automatically convert the JSON response into a Java Map so you can access it easily.

So basically:

- You send the user message to Gemini AI.
- Gemini AI processes it and returns a response.
- That response is stored in response.getBody(), which is then parsed to extract the text the bot should reply with.

It's like the bridge between your Java bot and Gemini AI.

Below code is parsing the response from Gemini and deciding what to send back to the user.

```
if (body != null && body.containsKey("candidates")) {
                        List<Map<String, Object>> candidates =
(List<Map<String, Object>>) body.get("candidates");
                        if (!candidates.isEmpty()) {
                          Map<String, Object> firstCandidate = candidates.get(0);
                          Map<String, Object> contentMap = (Map<String,
Object>) firstCandidate.get("content");

                          if (contentMap != null) {
                            List<Map<String, Object>> parts = (List<Map<String,
Object>>) contentMap.get("parts");

                            if (parts != null && !parts.isEmpty() &&
parts.get(0).containsKey("text")) {
                              return (String) parts.get(0).get("text");
```

```
                } else {
                    return "🤖 AI responded but didn't return any text
(finishReason="
                            + firstCandidate.get("finishReason") + ")";
                }
            }
        }
```

1. **Check if the response has a body and "candidates":**

if (body != null && body.containsKey("candidates")) { ... }

- The Gemini AI API returns a JSON with a field called "candidates" which contains one or more generated outputs.
- We first make sure the body exists and that it has "candidates".

2. **Get the list of candidates:**

List<Map<String, Object>> candidates = (List<Map<String, Object>>) body.get("candidates");

- Each candidate represents a possible AI response. Usually, we just take the first one.

3. **Pick the first candidate and get its content:**

Map<String, Object> firstCandidate = candidates.get(0);

Map<String, Object> contentMap = (Map<String, Object>) firstCandidate.get("content");

- Each candidate has a "content" field which contains the actual generated message parts.

4. **Get the list of parts inside content:**

List<Map<String, Object>> parts = (List<Map<String, Object>>) contentMap.get("parts");

- The AI response can have multiple "parts", e.g., text blocks or structured outputs.

5. **Check if parts contain text and return it:**

if (parts != null && !parts.isEmpty() && parts.get(0).containsKey("text")) {

return (String) parts.get(0).get("text");

}

- If the first part contains "text", we return it as the AI's response.

6. **Handle special cases:**

else {

return "🤖 AI responded but didn't return any text (finishReason="

+ firstCandidate.get("finishReason") + ")";

}

- If no text is returned but the AI response exists, it gives a fallback message showing the "finishReason" (like MAX_TOKENS if it stopped because of token limit).

7. **Catch exceptions:**

} catch (Exception e) {

e.printStackTrace();

return "🤖 AI is unavailable right now. Error: " + e.getMessage();

}

- If anything goes wrong (network issues, parsing errors), it catches the exception and returns a friendly error.

8. **Final fallback:**

return "🤖 AI returned an empty or invalid response.";

- If none of the above conditions are satisfied, it returns this default message.

✅ **In short:**

- This code **parses the AI's JSON response**, safely extracts the generated text if it exists, handles cases where no text is returned, and deals with exceptions gracefully.

package com.chatBot.service;

import java.util.Collections;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.http.HttpEntity;

import org.springframework.http.HttpHeaders;

import org.springframework.http.MediaType;

import org.springframework.http.ResponseEntity;

```
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

import com.chatBot.config.GoogleApiConfig;
import com.fasterxml.jackson.databind.ObjectMapper;

@Service
public class WhatsAppService {

        private final GoogleApiConfig googleApiConfig;

        public WhatsAppService(GoogleApiConfig googleApiConfig) {
    this.googleApiConfig = googleApiConfig;
  }

        @Value("${whatsapp.phoneNumberId}")
        private String phoneNumberId;

         @Value("${whatsapp.accessToken}")
         private String accessToken;

         private Map<String, String> userStates = new HashMap<>();
         private Map<String, Map<String, String>> userData = new HashMap<>();
         private RestTemplate restTemplate= new RestTemplate();
         /*
          * RestTemplate is a Spring class used to send HTTP requests and receive HTTP
responses from another server.
          *It lets your Spring Boot application act like a client calling APIs.
          */
public void sendMessage(String toPhone, String messageText)
```

```
{
        /*
         * This is the WhatsApp Cloud API endpoint provided by Meta (Facebook).
         *It's used to send messages from your WhatsApp Business account (or sandbox test number) to a user.
         * The {phone-number-id} is a unique identifier for your WhatsApp Business phone number.
         * You get this ID when you set up your WhatsApp Business account and link a phone number to it.
         *
         * The structure is:
https://graph.facebook.com/<API_VERSION>/<PHONE_NUMBER_ID>/messages
Where:
<API_VERSION> → version of the Graph API you're using, e.g., v17.0
<PHONE_NUMBER_ID> → your WhatsApp bot number's ID (the "From" number)
/messages → endpoint to send a message
         *
         */
 String url="https://graph.facebook.com/v17.0/" + phoneNumberId + "/messages";

//Creates a container for HTTP headers.HTTP headers carry metadata for the request, like authorization and content type.
HttpHeaders headers= new HttpHeaders();

//Sets Authorization header using Bearer token. WhatsApp Cloud API requires an access token to verify that your bot is allowed to send messages.
headers.setBearerAuth(accessToken);

//Tells the API that we are sending JSON data in the request body. WhatsApp Cloud API expects message data as JSON.
headers.setContentType(MediaType.APPLICATION_JSON);
```

```
//This builds the actual message body we send to WhatsApp.
/*
 * "messaging_product": "whatsapp" → API knows this is a WhatsApp message.
                "to": toPhone → Recipient phone number (user who sent the message).
                "type": "text" → Type of message (text, image, etc.).
                "text": {"body": messageText} → The actual text content of the message.
 */
Map<String, Object> payload = Map.of(
        "messaging_product", "whatsapp",
        "to", toPhone,
        "type", "text",
        "text", Map.of("body", messageText)
    );
//Wrapping Payload + Headers into an HttpEntity object.
/*HttpEntity bundles:
                Body → the JSON payload
                Headers → Authorization & Content-Type
                This is what RestTemplate sends as a full HTTP request.
*/
HttpEntity<Map<String, Object>> request = new HttpEntity<>(payload, headers);
//Sending the Request
restTemplate.postForEntity(url, request, String.class);
}
public void handleIncomingMessage(Map<String, Object> payload) {
try {
        List<Map<String, Object>> entry=(List<Map<String, Object>>)payload.get("entry");
        List<Map<String, Object>> changes = (List<Map<String, Object>>)
entry.get(0).get("changes");
        Map<String, Object> value = (Map<String, Object>) changes.get(0).get("value");
        List<Map<String, Object>> messages = (List<Map<String, Object>>)
value.get("messages");
```

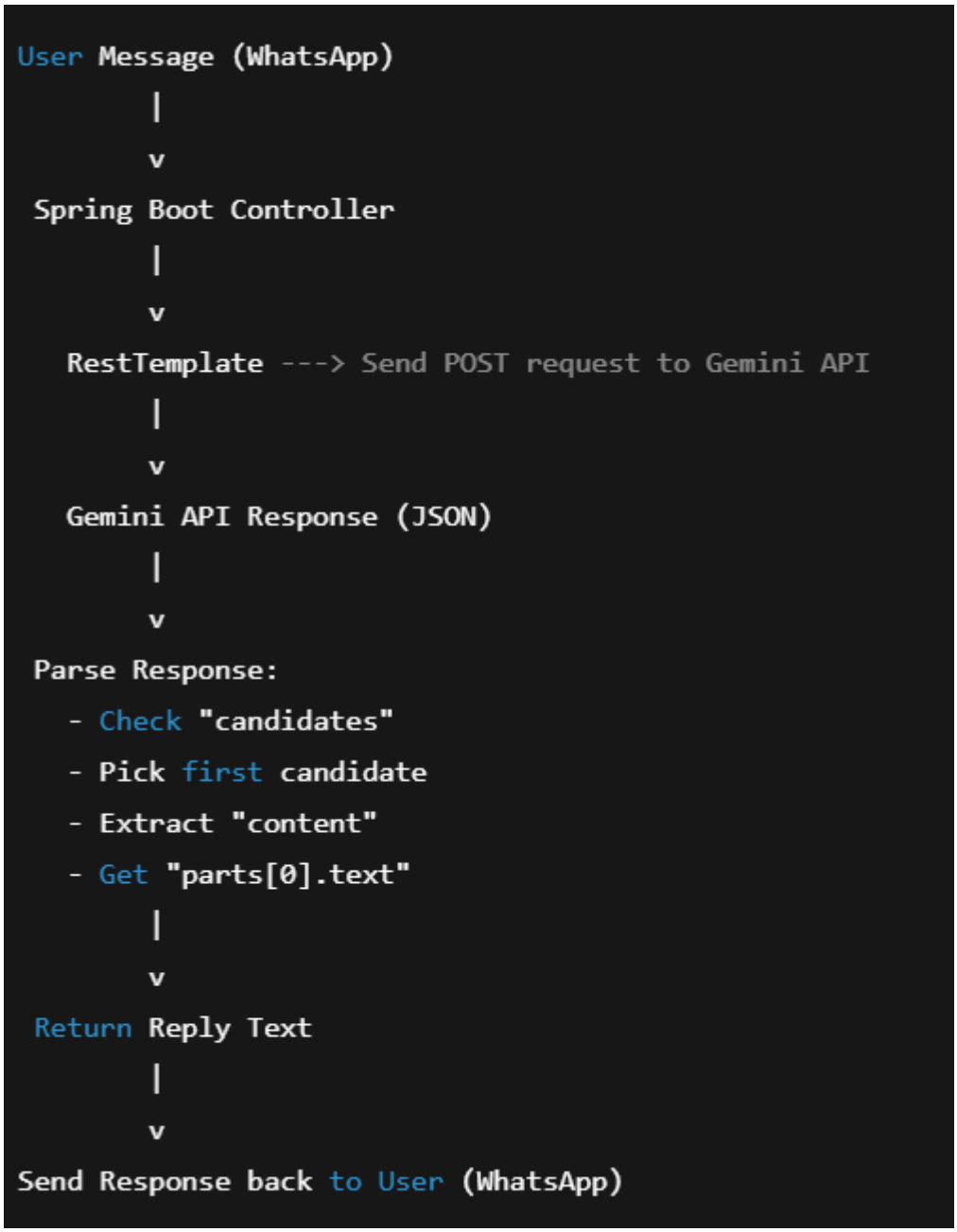
```
if(messages !=null && !messages.isEmpty())
{
        Map<String, Object>message=messages.get(0);
        String userPhone=(String)message.get("from");
        String text = ((Map<String, Object>) message.get("text")).get("body").toString();

        userData.putIfAbsent(userPhone, new HashMap<>());
        String state = userStates.getOrDefault(userPhone, "INIT");

        switch(state) {

        case "INIT":
        if(text.toLowerCase().contains("order")) {
            sendMessage(userPhone, "Hello! Welcome to our restaurant. May I know
your name?");
            userStates.put(userPhone, "ASK_NAME");
          } else {
            String aiReply = getGeminiAIResponse(text);
            sendMessage(userPhone, aiReply);
          }
          break;

    case "ASK_NAME":
      userData.get(userPhone).put("name", text);
      sendMessage(userPhone, "Thanks " + text + "! Here's our menu:\n1. Pizza\n2.
Burger\n3. Pasta\nPlease type your choice.");
      userStates.put(userPhone, "TAKE_ORDER");
      break;

    case "TAKE_ORDER":
      userData.get(userPhone).put("order", text);
```

```
            sendMessage(userPhone, "Great! How would you like to pay? (Cash / UPI / Card)");
            userStates.put(userPhone, "ASK_PAYMENT");
            break;

        case "ASK_PAYMENT":
            userData.get(userPhone).put("payment", text);
            String name = userData.get(userPhone).get("name");
            String order = userData.get(userPhone).get("order");
            String payment = userData.get(userPhone).get("payment");
            sendMessage(userPhone, "Thank you, " + name + ". Your order: " + order + " will be processed. Payment method: " + payment + ". Enjoy your meal! 🍽️");
            userStates.remove(userPhone);
            userData.remove(userPhone);
            break;

        default:
            String aiReply = getGeminiAIResponse(text);
            sendMessage(userPhone, aiReply);
            break;
    }
        }
            }
            catch(Exception e)
            {
                e.printStackTrace();
            }
        }
    private String getGeminiAIResponse(String userMessage) {
            try {
                String apikey = googleApiConfig.getApiKey();
```

```
                System.out.println("---Connecting to Gemini AI...");


                // Correct Gemini endpoint
                String url =
"https://generativelanguage.googleapis.com/v1beta/models/gemini-2.5-
flash:generateContent?key=" + apikey;
                HttpHeaders headers = new HttpHeaders();
                headers.setContentType(MediaType.APPLICATION_JSON);


                // Request body
                Map<String, Object> textPart = Map.of("text", userMessage);
                Map<String, Object> content = Map.of("parts",
Collections.singletonList(textPart));


                Map<String, Object> requestBody = new HashMap<>();
                requestBody.put("contents", Collections.singletonList(content));
                requestBody.put("generationConfig", Map.of(
                    "temperature", 0.7,
                    "maxOutputTokens", 1000
                ));


                HttpEntity<Map<String, Object>> request = new
HttpEntity<>(requestBody, headers);


                //  ◇ Debug: print request body
                System.out.println(">>> Gemini Request JSON: " + new
ObjectMapper().writeValueAsString(requestBody));
                ResponseEntity<Map> response = restTemplate.postForEntity(url,
request, Map.class);


//  ◇ print raw response
   System.out.println(">>> Gemini Raw Response: " + response);
```

```
    Map<String, Object> body = response.getBody();
    if (body != null && body.containsKey("candidates")) {
            List<Map<String, Object>> candidates = (List<Map<String, Object>>)
body.get("candidates");
            if (!candidates.isEmpty()) {
                    Map<String, Object> firstCandidate = candidates.get(0);
                    Map<String, Object> contentMap = (Map<String, Object>)
firstCandidate.get("content");

                    if (contentMap != null) {
                            List<Map<String, Object>> parts = (List<Map<String, Object>>)
contentMap.get("parts");
                            if (parts != null && !parts.isEmpty() &&
parts.get(0).containsKey("text")) {
                               return (String) parts.get(0).get("text");
                            } else {
                               return "🤖 AI responded but didn't return any text
(finishReason="
                                       + firstCandidate.get("finishReason") + ")";
                            }
                        }
                    }
                }
            } catch (Exception e) {
               e.printStackTrace();
               return "🤖 AI is unavailable right now. Error: " + e.getMessage();
            }
            return "🤖 AI returned an empty or invalid response.";
        }
}
```

There might be update in the code. You can see the complete code in github



🧠 UserSession — Managing Per-User Conversation Data

🔹 What is UserSession?

UserSession is a simple Java model class used to **store temporary, per-user data** during an active WhatsApp conversation.
It represents the **current state of a user's order session**, including:

- Customer's name

- Items they've ordered
- Prices of each item
- Total bill amount
- Chosen payment method

```
package com.chatBot.model;

import java.util.ArrayList;
import java.util.List;
import lombok.Data;

@Data
public class UserSession {
   private String name;
   private List<String> orderedItems = new ArrayList<>();
   private List<Double> prices = new ArrayList<>();
   private String payment;
   private Double total = 0.0;
}
```

🔹 **Why we added UserSession**

Before introducing UserSession, we used:

private Map<String, Map<String, String>> userData;

This was fine for storing simple values (like name or payment type),
but failed when we needed to handle **complex data structures** such as:

- Multiple ordered items per user
- List of item prices
- Calculating total bill dynamically

Since the inner map only accepted String values, we couldn't store List or Double types properly.
That's why we replaced it with a **typed object-based session** system.

🔹 **How it works**

1. **When a user starts chatting**, a new UserSession object is created for their phone number:

   userSessions.putIfAbsent(userPhone, new UserSession());

2. **During the chat**, data is added step-by-step:
   - In *ASK_NAME*, user's name is saved
   - In *TAKE_ORDER*, each selected menu item is added

- When the user types done, total bill is calculated
- In *ASK_PAYMENT*, payment choice and order summary are finalized

3. **Once the order is complete**, the session is removed:
   `userSessions.remove(userPhone);`

🔹 **Key Benefits**

✅ **Type safety** — avoids unsafe casts (no need for Map<String, Object>).

✅ **Easier to extend** — new fields (like delivery address, order ID, etc.) can be added anytime.

✅ **Cleaner code** — no confusing nested map lookups.

✅ **Per-user tracking** — each user has a separate active session.

✅ **Scalable for future DB integration** — when saving final orders to the database, we can easily convert a UserSession to an Order entity.

Step	User Action	Bot Response	Data Stored in UserSession
1	User types “order”	Bot asks for name	—
2	User enters name	Bot shows menu	name
3	User selects multiple items	Bot confirms each addition	orderedItems, prices
4	User types “done”	Bot shows total & asks payment	total
5	User gives payment method	Bot confirms order	payment, final total
6	Order complete	Session cleared	—

🔹 What It Stores

Each UserSession object holds:

- name → customer’s name
- orderedItems → list of all food items selected
- prices → prices of each selected item
- total → total bill amount (sum of all items)
- payment → payment method (Cash / UPI / Card)

🔹 **Summary**

UserSession = Smart in-memory container for tracking each user's ongoing chat state and order details.

It makes the chatbot **robust, extendable, and easier to maintain** — especially when handling multiple simultaneous users.

```
package com.chatBot.model;

import java.util.ArrayList;

import java.util.List;

import lombok.Data;


@Data

public class UserSession {

   private String name;

   private List<String> orderedItems = new ArrayList<>();

   private List<Double> prices = new ArrayList<>();

   private String payment;

   private Double total = 0.0;

}
```

```
// To store active user sessions temporarily in memory
// Key → user's phone number, Value → session details (name, items, etc.)
private Map<String, UserSession> userSessions = new ConcurrentHashMap<>();

// When user enters name
userSessions.putIfAbsent(userPhone, new UserSession()); // create new session if not present
UserSession session = userSessions.get(userPhone());   // get session for current user
session.setName(text);                        // store user name

// When user selects items
session.getOrderedItems().add(selectedItem.getName()); // add selected item name
session.getPrices().add(selectedItem.getPrice());     // store its price

// When user finishes order
double total = session.getPrices().stream()
            .mapToDouble(Double::doubleValue)
            .sum();                  // calculate total bill
session.setTotal(total);                        // store total bill

// When user provides payment
```

```
session.setPayment(paymentMethod);          // store payment method (Cash/UPI/Card)

// After order confirmation
userSessions.remove(userPhone);             // clear session after completion
```

Storing the order in DB

```
package com.chatBot.model;

import java.time.LocalDateTime;

import java.util.List;

import jakarta.persistence.CascadeType;

import jakarta.persistence.Entity;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.GenerationType;

import jakarta.persistence.Id;

import jakarta.persistence.OneToMany;

import jakarta.persistence.Table;

import lombok.Data;


@Entity

@Data

@Table(name = "orders")

public class Order {


    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;


    private String customerName;

    private String userPhone;

    private String status="PENDING"; //PENDING, CONFIRMED, COMPLETED

    private String paymentMode;

    private Double totalPrice;
```

private LocalDateTime orderTime;

@OneToMany(mappedBy = "order", cascade = CascadeType.ALL)

private List<OrderItem>orderItems;

}

✅ **Order**

Purpose:
Represents a full order placed by the user. It connects customer details, payment info, and all ordered items.

Field	Type	Purpose
id	Long	Primary key — uniquely identifies each order.
customerName	String	Stores the name of the customer who placed the order.
userPhone	String	Stores customer's contact number (can be used to identify user).
status	String	Tracks the current order status: "PENDING", "CONFIRMED", or "COMPLETED".
paymentMode	String	Type of payment: Cash / UPI / Card.
totalPrice	Double	Stores total bill amount for all items.
orderTime	LocalDateTime	Timestamp when the order was placed.
orderItems	List<OrderItem>	Links all items (with quantity) in the order. Defined using @OneToMany(mappedBy = "order", cascade = CascadeType.ALL).

💡 **Why we added this:**
To save one complete order per customer, including total price, payment type, and status, which helps the restaurant owner manage all incoming orders easily.

✅ **MenuItem**

package com.chatBot.model;

import jakarta.persistence.Column;

import jakarta.persistence.Entity;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.GenerationType;

import jakarta.persistence.Id;

import jakarta.persistence.Table;

import lombok.AllArgsConstructor;

```
import lombok.Data;
import lombok.NoArgsConstructor;

/*
 * MenuItem entity representing a menu item in the restaurant.
 * If the table is not present, it will be created automatically with the class name.
 */
@Entity
@Table(name = "menu_items")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class MenuItem {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(length = 1000)
    private String description;

    @Column(nullable = false)
    private Double price;

    @Column(nullable = false)
    private boolean available;
```

}

Purpose:
Represents individual dishes available in the restaurant menu.

Field	Type	Purpose
id	Long	Primary key for menu item.
name	String	Name of the dish (e.g., Biryani, Paneer Tikka).
description	String	Optional short description of the dish.
price	Double	Price of one unit of the dish.
available	boolean	Tells whether the dish is available or out of stock.

💡 **Why we added this:**
The restaurant's master menu list — every order picks items from here.

✅ **OrderItem**

Purpose:
Joins Order and MenuItem to store **which items** and **how many quantities** are included in a particular order.

```
package com.chatBot.model;

import jakarta.persistence.Entity;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.GenerationType;

import jakarta.persistence.Id;

import jakarta.persistence.JoinColumn;

import jakarta.persistence.ManyToOne;

import lombok.Data;


@Entity

@Data

public class OrderItem {


    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private Long id;

    @ManyToOne
  @JoinColumn(name = "order_id")
    private Order order;

    @ManyToOne
  @JoinColumn(name = "menu_item_id")
    private MenuItem menuItem;
    private int quantity;
}
```

Field	Type	Purpose
id	Long	Unique identifier for each order item record.
order	Order	The parent order — linked with @ManyToOne to Order.
menuItem	MenuItem	The menu item that was ordered.
quantity	int	Quantity of that particular menu item in this order.

💡 **Why we added this:**
This helps us store multiple items per order and calculate total bills dynamically (e.g., 2x Biryani + 1x Paneer Tikka).

📘 **Repository Layer Explanation (for Documentation)**

✅ **MenuItemRepository**

Purpose:
To interact with the menu_items table — handles all operations related to restaurant menu data.

```
package com.chatBot.repository;

import java.util.List;

import java.util.Optional;

import org.springframework.data.jpa.repository.JpaRepository;

import com.chatBot.model.MenuItem;

public interface MenuItemRepository extends JpaRepository<MenuItem, Long> {
```

List<MenuItem> findByAvailableTrue();

Optional<MenuItem> findByNameIgnoreCase(String name);

}

Key methods:

Method	Purpose
findByAvailableTrue()	Fetches only available dishes (menu items where available = true).
findByNameIgnoreCase(String name)	Finds a menu item by name (case-insensitive). Useful when users type names like “biryani” or “BIRYANI”.

✅ OrderRepository

Purpose:
To perform CRUD operations on the orders table — where each user order is stored.

package com.chatBot.repository;

import org.springframework.data.jpa.repository.JpaRepository;

import com.chatBot.model.Order;

public interface OrderRepository extends JpaRepository<Order, Long>{

}

Use cases:

- Save new orders.
- Retrieve all orders for dashboard view.
- Update order status (Confirm / Cancel / Complete).

✅ OrderItemRepository

Purpose:
Handles data persistence for individual items within an order (order_items table).
This is usually managed automatically via cascading when saving an Order.

package com.chatBot.repository;

import org.springframework.data.jpa.repository.JpaRepository;

import com.chatBot.model.OrderItem;

public interface OrderItemRepository extends JpaRepository<OrderItem, Long> {

}

⚙️ **Next Step (Business Logic Layer)**

Now that your repositories are set up, the next step is to create a **Service class** to:

1. Build the order from userSession data.
2. Create corresponding OrderItem objects.
3. Calculate total price.
4. Save everything to the database using OrderRepository.

```
package com.chatBot.service;

import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.chatBot.model.MenuItem;
import com.chatBot.model.Order;
import com.chatBot.model.OrderItem;
import com.chatBot.repository.MenuItemRepository;
import com.chatBot.repository.OrderItemRepository;
import com.chatBot.repository.OrderRepository;

@Service
public class OrderService {
        @Autowired
  private OrderRepository orderRepository;

  @Autowired
  private MenuItemRepository menuItemRepository;

  @Autowired
  private OrderItemRepository orderItemRepository;

  /**
   * Saves a new order in the database.
   *
   * @param customerName  - Name of the customer
   * @param userPhone     - Customer's phone number
   * @param paymentMode   - Selected payment mode (Cash / UPI / Card)
   * @param sessionItems  - Map of <itemName, quantity> from user session
   * @return Saved Order entity with generated ID
   */

  public Order saveOrder(String customerName, String userPhone, String
paymentMode, Map<String, Integer> sessionItems) {
```

```
        Order order= new Order();
        order.setCustomerName(customerName);
        order.setUserPhone(userPhone);
        order.setPaymentMode(paymentMode);
        order.setOrderTime(LocalDateTime.now());
        order.setStatus("PENDING");

        List<OrderItem> orderItemList= new ArrayList<>();
        double total=0.0;

        for(Map.Entry<String,Integer>entry:sessionItems.entrySet())
        {
                String itemName=entry.getKey();
                int quantity=entry.getValue();


                //Find MenuItem by name
                MenuItem
menuItem=menuItemRepository.findByNameIgnoreCase(itemName)
                                        .orElseThrow(()->new RuntimeException("Menu
item not found: " + itemName));


                //Create OrderItem
                OrderItem orderItem= new OrderItem();
                orderItem.setOrder(order);
                orderItem.setMenuItem(menuItem);
                orderItem.setQuantity(quantity);

                //Calculate total
                total+=menuItem.getPrice()*quantity;
                orderItemList.add(orderItem);
        }

        order.setOrderItems(orderItemList);
        order.setTotalPrice(total);

        //Save Order (which will also save OrderItems due to CascadeType.ALL)
        Order savedOrder= orderRepository.save(order);

        return savedOrder;

    }
@Transactional
    public Order getOrderById(Long orderId) {
        return orderRepository.findById(orderId)
                .orElseThrow(() -> new RuntimeException("Order not found: " + orderId));
    }

    @Transactional
    public Order updateOrder(Order order) {
```

```
    return orderRepository.save(order);
  }



}
```

let's integrate this now — so when the user **confirms their order and chooses a payment mode (Cash / UPI / Card)**, the system automatically saves the order to the database.

now **integrate the DB saving logic** into your existing WhatsAppService without breaking the flow.

1️⃣ Inject OrderService in WhatsAppService

Add at the top of your class:

@Autowired

private OrderService orderService;

OrderService will handle saving the Order + OrderItems to DB.

2️⃣ Update ASK_PAYMENT case to save order

Replace the // TODO: save Order + OrderItems into DB here section with:

After the below

"

String confirmation = "✅ Thank you, " + session.getName() + ".\n" +

"Your order:\n" +

session.getOrderedItems().stream().map(i -> "• " + i).collect(Collectors.joining("\n")) +

"\n 💰 Total Bill: ₹" + total +

"\nPayment method: " + session.getPayment() +

"\n\nYour meal will be ready soon! 🍽️";

sendMessage(userPhone, confirmation);

"

💳 Razorpay Payment Integration

🎯 Objective

Integrate a secure, automated, and user-friendly online payment system into the **Restaurant Chatbot Application**, allowing customers to complete their orders seamlessly via Razorpay Payment Gateway.

🧠 Why Chosed Razorpay

1. **Ease of Integration** – Provides a clean Java SDK and REST APIs easily usable with Spring Boot.
2. **Multiple Payment Options** – UPI, Credit/Debit Cards, Net Banking, Wallets.
3. **Webhook Support** – Allows real-time tracking of payment status through server callbacks.
4. **Developer-Friendly Documentation** – Well-documented API with minimal setup.
5. **Sandbox Mode** – Enables testing without using real money.

⚙️ Implementation Flow

🧩 Step 1: Razorpay Account Setup

1. Create a Razorpay account.
2. Generate **Key ID** and **Key Secret** from Razorpay Dashboard → Settings → API Keys.
3. Store these credentials securely in **application.properties** (Spring Boot):

```
# Razorpay Credentials
razorpay.keyId=rzp_test_XXXXXXXXXXXXXXX
razorpay.keySecret=XXXXXXXXXXXXXXXXXXX
```

4. Create a configuration class in Spring Boot to map these properties:

```
@Configuration
@ConfigurationProperties(prefix = "razorpay")
@Data
public class RazorpayConfig {
    private String keyId;
    private String keySecret;
}
```

✅ This ensures that credentials are injected securely into your application wherever needed.

💳 **Step 2: Generating Dynamic Payment Links**

Razorpay **requires the amount to be specified in paisa**, not rupees.
So, always multiply the order amount by 100 before creating the payment link.

When a user confirms an order via the chatbot:

1. **Backend generates a unique payment link** using RazorpayService.
2. Attach user-specific information and order details:

```
double orderAmountInRupees = 499.50; // ₹499.50
int amountInPaisa = (int) (orderAmountInRupees * 100); // 49950 paisa

Map<String, Object> request = new HashMap<>();
request.put("amount", amountInPaisa); // Razorpay expects amount in paisa
request.put("currency", "INR");
request.put("reference_id", "order_refid_" + orderId);
request.put("notes", new JSONObject()
    .put("order_type", "WhatsApp Bot")
    .put("ResturantOrder_ID", "order_refid_" + orderId));
```

3. Razorpay returns a short payment URL: *https://rzp.io/i/abc123*

✅ **Result:**

Razorpay generates a short payment URL (e.g. https://rzp.io/i/abc123), which is sent to the customer on WhatsApp to complete payment.

🌐 **Step 3: Adding Webhook URL in Razorpay Dashboard**

To enable automatic payment confirmation:

1. Go to your **Razorpay Dashboard** → **Settings** → **Webhooks**.
2. Click **"+ Add New Webhook"**.
3. Enter your **public webhook URL**, for example:
   https://your-ngrok-url.ngrok-free.dev/api/payment/callback
4. Choose the following **event types** (minimum required):
    1. payment_link.paid
    2. payment_link.failed
    3. payment.captured
    4. payment.failed

5. Save the webhook.

   ⚠️ **Important:**

   Razorpay webhooks won't work with localhost.

   Use **ngrok** or deploy your app online to get a public URL (e.g., https://xxxxx.ngrok-free.dev).

📬 **Step 4: Handling Razorpay Webhook in Spring Boot**

- Razorpay sends a **POST request** to /api/payment/callback after payment events.
- The backend extracts the internal order ID from notes.ResturantOrder_ID and updates the order status automatically.

Example Implementation:

```
if ("paid".equalsIgnoreCase(paymentStatus) || "captured".equalsIgnoreCase(paymentStatus)) {
    order.setStatus("CONFIRMED");
    orderService.updateOrder(order);
    whatsAppService.sendMessage(order.getUserPhone(),
        "✅ Payment received successfully!\nYour order (ID: " + orderId + ") is confirmed.");
}
else if ("failed".equalsIgnoreCase(paymentStatus)) {
    order.setStatus("PAYMENT_FAILED");
    orderService.updateOrder(order);
    whatsAppService.sendMessage(order.getUserPhone(),
        "❌ Payment failed for order ID: " + orderId + ". Please try again.");
}
```

✅ **Outcome:**

- Payment success → Order status = *CONFIRMED*
- Payment failure → Order status = *PAYMENT_FAILED*
- Customer gets instant WhatsApp message about payment result.

🧾 **Step 5: Manual Payment Verification (Optional)**

If you need to verify payments manually:

- Open **Razorpay Dashboard → Payments → Transaction Details**.
- Check the **Notes** section for the internal order reference:
  ResturantOrder_ID: order_refid_41

  Match this ID with your local database record to confirm payment.

```
package com.chatBot.service;

import org.json.JSONObject;
import org.springframework.stereotype.Service;
```

```
import com.chatBot.config.RazorpayConfig;
import com.razorpay.PaymentLink;
import com.razorpay.RazorpayClient;
import com.razorpay.RazorpayException;

@Service
public class RazorpayService {

	private final String keyId;
	private final String keySecret;

	public RazorpayService(RazorpayConfig config)
	{
		this.keyId=config.getKeyId();
		this.keySecret=config.getKeySecret();
	}

	 /**
    * Creates a Razorpay payment link for a given order.
    *
    * @param orderId       - Internal order ID (used for dynamic receipt)
    * @param customerName  - Customer name
    * @param customerEmail  - Customer email
    * @param customerPhone  - Customer phone
    * @param amount        - Amount to be paid
    * @return short_url of the Razorpay payment link
    * @throws RazorpayException
    */
	public String createPaymentLink(Long orderId,String customerName,
String customerEmail, String customerPhone, Double amount) throws
RazorpayException
	{
		RazorpayClient client= new RazorpayClient(keyId,keySecret);
		JSONObject request= new JSONObject();
		request.put("amount",(int)(amount*100)); // amount in paisa
		request.put("currency","INR");
		request.put("accept_partial", false);

		 JSONObject customer = new JSONObject();
	    customer.put("name", customerName);
	    customer.put("email", customerEmail);
	    customer.put("contact", customerPhone);

	    request.put("customer", customer);
	    request.put("notify", new JSONObject().put("sms",
true).put("email", true));
```

```
            request.put("reminder_enable", true);
            request.put("reference_id", "order_refid_" + orderId);
                request.put("notes", new JSONObject().put("order_type",
"WhatsApp Bot").put("ResturantOrder_ID", "order_refid_" + orderId));

          // ☑ create returns PaymentLink object
            PaymentLink paymentLink=client.paymentLink.create(request);

          // ☑ Convert to JSONObject if you want to access data
            JSONObject response=paymentLink.toJson();

          // ☑ Extract short_url field
            return response.getString("short_url");
        }


        public void cancelPaymentLink(String paymentLinkId) {
            try {
                RazorpayClient client = new RazorpayClient(keyId, keySecret);
                client.paymentLink.cancel(paymentLinkId);
                System.out.println("☑ Payment link cancelled: " +
paymentLinkId);
            } catch (Exception e) {
                e.printStackTrace();
                System.out.println("⚠ Error cancelling payment link: " +
e.getMessage());
            }
        }



}
```

```
package com.chatBot.controller;

import java.util.Map;
import org.json.JSONObject;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import com.chatBot.model.Order;
```

```
import com.chatBot.service.OrderService;
import com.chatBot.service.RazorpayService;
import com.chatBot.service.WhatsAppService;

@RestController
@RequestMapping("/api/payment")
public class RazorpayController {

    // Services for order management and sending WhatsApp messages
    private final OrderService orderService;
    private final WhatsAppService whatsAppService;
    private final RazorpayService razorpayService;

    // Constructor injection for the required services
    public RazorpayController(OrderService orderService, WhatsAppService
whatsAppService,RazorpayService razorpayService) {
        this.orderService = orderService;
        this.whatsAppService = whatsAppService;
        this.razorpayService=razorpayService;
    }

    /**
     * Handles Razorpay payment callbacks for both successful and failed
payments.
     * The payload is sent by Razorpay when payment_link or payment events
occur.
     *
     * @param payload - JSON payload sent by Razorpay webhook
     * @return ResponseEntity containing success/failure messages or error
info
     */


@PostMapping("/callback")
    public ResponseEntity<?> handlePaymentCallback(@RequestBody
Map<String, Object> payload) {
        try {
            // Log the incoming payload for debugging
            System.out.println("Razorpay Callback Payload: " + payload);

            // Extract the 'payload' object from Razorpay event
            Map<String, Object> innerPayload = (Map<String, Object>)
payload.get("payload");
            if (innerPayload == null) {
                return ResponseEntity.badRequest().body(Map.of("error", "Missing
inner payload"));
```

```
        }

        // Initialize 'entity' which contains payment/payment_link details
        Map<String, Object> entity = null;

        // Check if it's a payment_link event
        if (innerPayload.containsKey("payment_link")) {
            Map<String, Object> paymentLink = (Map<String, Object>)
innerPayload.get("payment_link");
            entity = (Map<String, Object>) paymentLink.get("entity");
        }
        // Otherwise, check if it's a direct payment event
        else if (innerPayload.containsKey("payment")) {
            Map<String, Object> payment = (Map<String, Object>)
innerPayload.get("payment");
            entity = (Map<String, Object>) payment.get("entity");
        }

        if (entity == null) {
            return ResponseEntity.badRequest().body(Map.of("error", "Missing
entity in payload"));
        }

        // Extract 'notes' to fetch internal DB order ID
        Map<String, Object> notes = (Map<String, Object>)
entity.get("notes");
        if (notes == null || !notes.containsKey("ResturantOrder_ID")) {
            return ResponseEntity.badRequest().body(Map.of("error", "Missing
internal order ID in notes"));
        }

        // Retrieve internal order ID stored in notes
        String orderStr = (String) notes.get("ResturantOrder_ID"); // e.g.,
"order_refid_41"
        Long orderId = null;
        try {
            // Extract numeric part of the ID after the last underscore
            orderId = Long.valueOf(orderStr.split("_")[2]);
        } catch (NumberFormatException e) {
            return ResponseEntity.badRequest().body(Map.of("error", "Invalid
order ID format in notes"));
        }

        // Get payment status (paid, captured, failed, etc.)
        String paymentStatus = (String) entity.get("status");
        if (paymentStatus == null) {
```

```
            return ResponseEntity.badRequest().body(Map.of("error", "Missing
payment status in payload"));
        }

        // Fetch order details from DB using internal order ID
        Order order = orderService.getOrderById(orderId);

        // Handle payment success
        if ("paid".equalsIgnoreCase(paymentStatus) ||
"captured".equalsIgnoreCase(paymentStatus)) {
            System.out.println(">>> Payment successful for order ID: " +
orderId);

            // Update order status to CONFIRMED in DB
            order.setStatus("CONFIRMED");
            orderService.updateOrder(order);

            // Send WhatsApp confirmation message to the customer
            String confirmationMsg = "☑ Payment received successfully!\n"
                    + "Payment Mode: " + order.getPaymentMode()
                    + "\nThank you *" + order.getCustomerName() + "😄* \n"
                    + "Your order (ID: " + orderId + ") has been confirmed.\n\n"
                    + "Your order will be ready soon! 🍽";
            whatsAppService.sendMessage(order.getUserPhone(),
confirmationMsg);

            return ResponseEntity.ok(Map.of("message", "Payment successful,
order confirmed"));

        }
        // Handle payment failure
        else if ("failed".equalsIgnoreCase(paymentStatus))
        {
            System.out.println(">>> Payment failed for order ID: " + orderId);

            // Update order status to PAYMENT_FAILED in DB
            order.setStatus("PAYMENT_FAILED");
            orderService.updateOrder(order);

            // Notify customer about failed payment and next steps
            String failureMsg = "❌ Payment failed for your order (ID: " +
orderId + ").\n"
                    + "Please try to do payment again with the given payment
link or Start a new order by typing \"*order*\" and use *_Cash on
Delivery_*.";
            whatsAppService.sendMessage(order.getUserPhone(), failureMsg);
```

```
                return ResponseEntity.ok(Map.of("message", "Payment failed,
order not confirmed"));
            }


            // Handle other statuses (pending, created, etc.)
            else {
                System.out.println(">>> Payment pending or unknown status for
order ID: " + orderId);
                return ResponseEntity.ok(Map.of("message", "Payment status: " +
paymentStatus));
            }

        } catch (Exception e) {
            // Log stack trace for debugging
            e.printStackTrace();
            // Return internal server error with exception message
            return ResponseEntity.internalServerError().body(Map.of("error",
e.getMessage()));
        }
    }
}
```

🚧 Problems Faced & Solutions while implementing Payment

Problem	Description	Solution
1. Webhook not triggering locally	Razorpay requires a public URL.	Used **ngrok** to expose the local Spring Boot port and added that URL in Razorpay dashboard.
2. Order not updating after payment	Payload didn't contain internal order ID initially.	Added *notes.ResturantOrder_ID* and *reference_id* when creating the payment link.
3. Duplicate webhook events	Razorpay triggers both *payment_link.paid* and *payment.captured*.	Added logic to handle one update only based on *status*.

Problem	Description	Solution
4. Incorrect order confirmation on failed payments	Orders were confirmed even for failed payments.	Added strict status validation (*paid* or *captured* only).
5. No customer feedback after payment	User wasn't informed of success/failure.	Integrated **WhatsAppService** to send real-time notifications.

✅ **Final Result**

- End-to-end payment automation using **Razorpay Payment Links** and **Webhooks**.
- Secure, real-time order confirmation system.
- Instant WhatsApp updates to users for both success and failure.
- Fully tested in sandbox and ready for production use.

FRONTEND

🍽️ Frontend (React JS) – Features & Implementation Details

⚙️ Tech Stack & Libraries Used

Purpose	Library / Tool	Description
Core Framework	**React JS (Vite)**	Fast frontend framework for building responsive UIs.
Routing	**react-router-dom**	For navigation between pages (e.g., Dashboard, Menu, Chat, Orders).
HTTP Requests	**axios**	Used for calling Spring Boot REST APIs.
Real-Time Communication	**SockJS** + **STOMP.js**	Used to connect to Spring Boot WebSocket server for real-time updates.
Styling	**Tailwind CSS**	For modern, responsive, and reusable styling components.
State Management	**React Hooks** (`useState`, `useEffect`)	Manage and update application state efficiently.
UI Animations	**Framer Motion** *(optional)*	Smooth UI transitions (used in modals or button animations).
Icons	**Lucide React / Heroicons**	For dashboard and menu icons.
Audio Notification	**HTML5 Audio API**	Used to play sound automatically on receiving new order events.

💡 Key Features Implemented

🏠 1. Interactive Admin Dashboard

- Displays **all live and completed orders** fetched from the backend.
- Real-time **order notification** sound plays when a new order arrives.
- Displays key stats: Total Orders, Pending, Delivered, and Revenue.

🔔 2. Real-Time Order Updates (WebSocket Integration)

- Connected to Spring Boot backend using **STOMP over SockJS**.
- Subscribed to a WebSocket topic, e.g. /topic/orders.
- Whenever the backend broadcasts a new order event, the frontend instantly receives it **without any refresh**.
- On receiving a message:
    - The new order is added to the list dynamically.
    - A **notification sound** plays automatically.

Implementation Snippet:

🧻 3. Order Details Modal

- Clicking on any order opens a modal showing:
    - Customer name
    - Delivery address
    - Ordered items, quantity, price, and total
    - Order time (formatted in readable format)
- Modal designed using **Tailwind CSS**, responsive and scrollable for long orders.

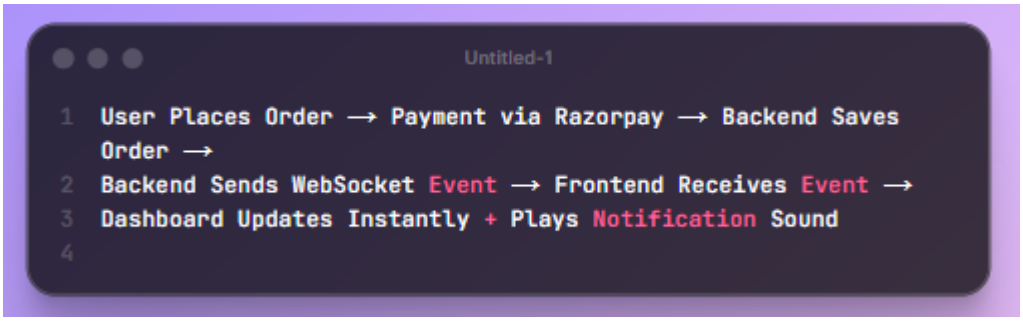
Item	Qty	Price
Paneer Biryani	2	₹180
Cold Drink	1	₹40

💳 4. Razorpay Payment Integration (Backend-Connected)

- When a customer makes a payment, Razorpay sends data to backend.
- Backend triggers WebSocket event → frontend dashboard instantly updates.
- This enables **real-time payment status** and **order confirmation** display.

🔄 How Real-Time Order Fetching Works

Flow Summary:



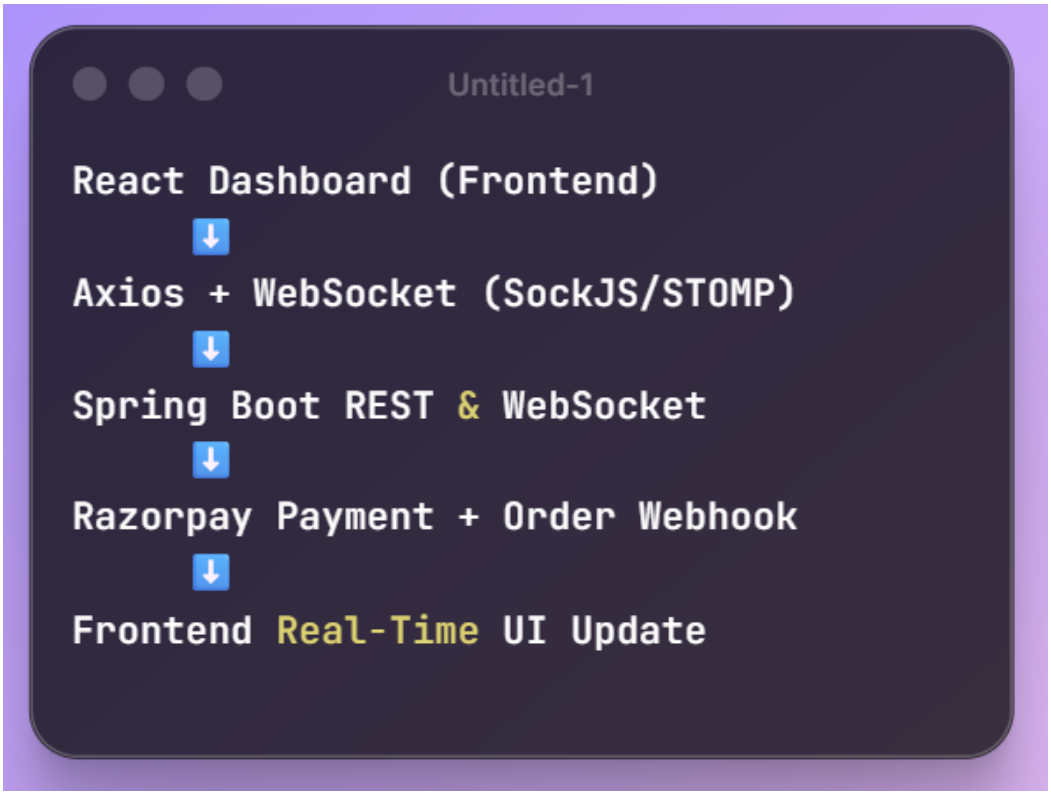
Step-by-Step Implementation:

1. **Backend** (Spring Boot):
    - Configured WebSocketConfig with @EnableWebSocketMessageBroker.
    - Backend publishes message to /topic/orders whenever a new order is created.
    - Added @SendTo in Controller or SimpMessagingTemplate.convertAndSend() for updates.
2. **Frontend** (React JS):
    - Established WebSocket connection on dashboard mount.
    - Subscribed to /topic/orders.
    - When message received:
        - Parsed JSON data.
        - Updated React state (setOrders).
        - Played notification sound via Audio.play().

Problems Faced & Solutions

Problem	Cause	Solution
Audio not playing automatically	Browser blocks autoplay	Played audio inside a user-triggered context or handled using useEffect after connection.
Orders not updating instantly	Missing STOMP subscription or wrong topic	Verified backend topic name and subscription path.
CORS/WebSocket connection error	Different frontend-backend ports	Enabled CORS in Spring Boot WebSocket config.
Payment webhook not triggering	Webhook not configured properly in Razorpay	Added correct /webhook URL in Razorpay dashboard and handled event in backend.

Overall Flow Summary



🧩 **What is STOMP?**

STOMP stands for **Simple Text Oriented Messaging Protocol**.
It is a **lightweight messaging protocol** that defines how clients and servers can communicate through messages — especially over **WebSockets**.

In short:
➡️ **WebSocket** provides the real-time connection (the "pipe").
➡️ **STOMP** defines the **format and rules** for sending messages through that pipe.

⚙️ How STOMP Works (Conceptually)

When you use WebSockets directly, you must manually send and handle raw data frames (like strings or JSON).
STOMP makes this process **structured and easier** by introducing:

- **Commands** (CONNECT, SUBSCRIBE, SEND, DISCONNECT)
- **Headers** (destination topics, message IDs)
- **Message bodies** (actual payloads like your order data)

Example STOMP Message:

This structured format allows multiple clients to **subscribe** to topics (like /topic/orders) and receive messages instantly when the server broadcasts them.

🏗️ How STOMP Fits into Your Project

🔸 **Backend (Spring Boot)**

- Spring Boot uses **@EnableWebSocketMessageBroker** and built-in STOMP support.

- When a new order is created, the backend **sends a STOMP message** to /topic/orders.

Example:

🔸 **Frontend (React JS)**

- You use **SockJS** (to connect to the WebSocket endpoint) and **STOMP.js** (to speak STOMP protocol over that connection).

So, when the backend sends a message, the STOMP client instantly receives it — no need to refresh or poll the API.

⚡ **Why We Use STOMP**

Problem	Solution via STOMP
Need real-time updates (orders, payments)	**STOMP over WebSocket delivers instantly.**
Want structured messaging (not raw sockets)	**STOMP defines clear send/subscribe protocol.**

Support multiple clients/topics	**STOMP supports topic-based broadcasting easily.**
Integrate smoothly with Spring Boot	**Spring has built-in STOMP broker support.**

🧠 Simple Analogy

Think of:

- **WebSocket as the telephone line that keeps two people connected.**
- **STOMP as the language or rules they follow while talking (e.g., one person speaks, the other listens to a specific topic).**

⚡ Real-Time Communication: WebSocket + STOMP in Your Project

🧠 1. What is WebSocket?

WebSocket is a bi-directional communication channel between the client (browser) and server.
Unlike HTTP (which closes after each request), WebSocket stays open — allowing:

- Server → Client instant messages
- Client → Server instant updates

That's why it's perfect for real-time applications (like chat, live orders, notifications, etc.).

🔸 2. Why Use STOMP Over WebSocket

WebSocket alone sends only raw data (like JSON strings), which is hard to manage for multiple message types.

STOMP (Simple Text Oriented Messaging Protocol) adds a structured messaging format on top of WebSocket.
It defines:

- Destinations (topics): e.g. /topic/orders
- Message structure: header + body
- Subscribe and send actions

Together, they create a reliable publish–subscribe system.

🧩 3. Flow of Real-Time Orders

Here's how it works in your Restaurant Dashboard:

🖥️ Frontend (React + STOMP.js + SockJS)

1. A SockJS client connects to backend endpoint:
   http://localhost:8080/ws
2. A STOMP client runs over that WebSocket connection.
3. The client subscribes to a specific topic —
   e.g., /topic/orders
4. Whenever a new message (order) is broadcast from the server,
   it instantly appears on the dashboard without refresh.

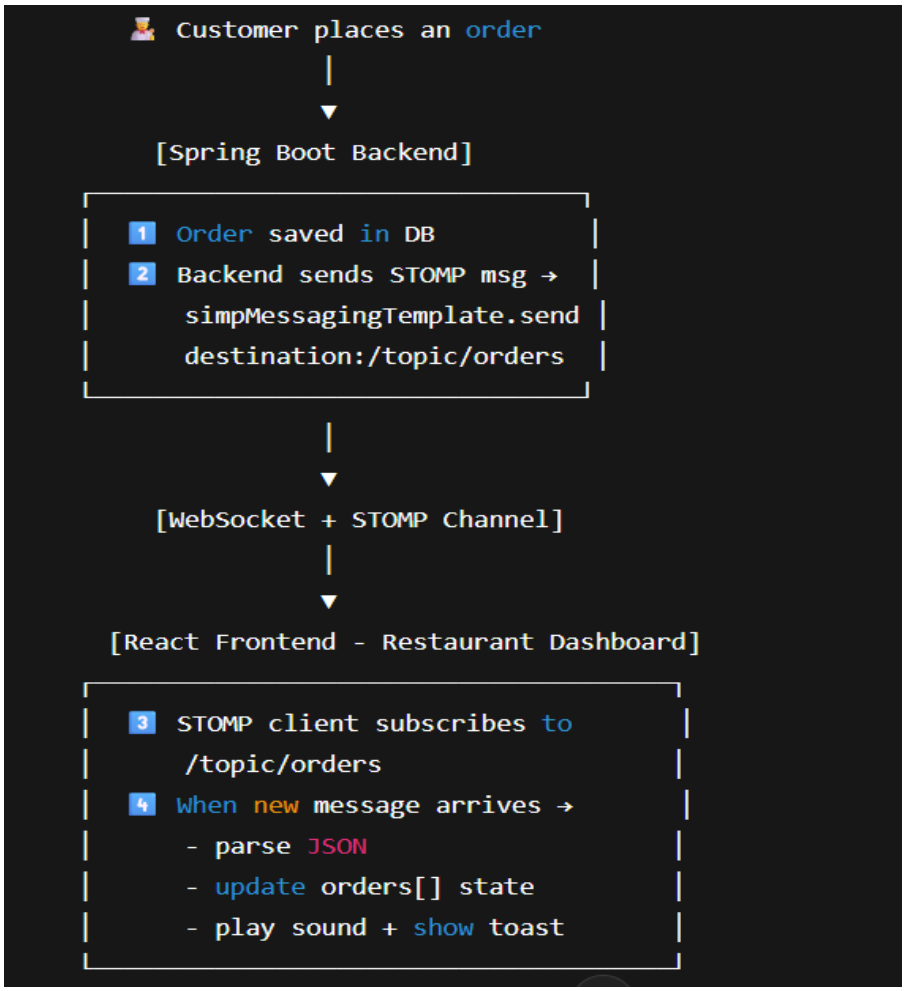
⚙️ Backend (Spring Boot)

1. When a new order is created or updated,
   backend triggers:

2. This sends the order to all subscribed clients.
3. The client receives the JSON payload and updates UI instantly.

🧭 4. Full Real-Time Data Flow Diagram

Here's the architecture flow 👇



Result:

🚀 **Instant live order updates** — no reloads, no API polling.

🧩 5. Libraries Used for Real-Time Orders

Library	Purpose
SockJS	Creates fallback-safe WebSocket connections (works even if browser blocks native WebSocket).
stompjs	Handles STOMP protocol — allows subscribe() and send() easily.
react-hot-toast	Shows toast notifications for new orders.
React Hooks (useEffect, useRef, useState)	Manage state, lifecycle, and audio playback.

⚙️ 6. How Frontend Code Works (Simplified)

✅ When the backend broadcasts,
every connected restaurant dashboard **instantly receives** the update.

🏁 Summary

Step	Description
1️⃣	Backend creates or updates an order.
2️⃣	Backend sends message to /topic/orders.
3️⃣	Frontend STOMP client receives it instantly.
4️⃣	UI updates + plays sound + toast shown.

💡 **That's how your system achieves true real-time order tracking — powered by WebSocket + STOMP.**

🧩 Real-Time Order Updates (WebSocket + STOMP Integration)

The restaurant dashboard implements **real-time order updates** using **WebSocket with STOMP protocol**.
The backend (Spring Boot) broadcasts new or updated orders using
SimpMessagingTemplate.convertAndSend("/topic/orders", order),
while the frontend (React) connects via **SockJS** and **stompjs**, subscribing to /topic/orders.

Whenever a new order is placed, the message is instantly pushed to all connected dashboards without any page reload or API polling. The UI dynamically updates the order list, plays a notification sound, and displays a toast alert.

This ensures seamless, instant synchronization between customers and the restaurant panel for live order tracking.

⚙️ Backend + Frontend Integration Flow (WebSocket + STOMP)

Your backend and frontend communicate through a **real-time WebSocket connection**, established using **STOMP protocol**.
Let's go through what each part does and how they work together 👇

🧩 1. Backend Setup

A. WebSocketConfig.java

✅ **What happens here:**

- A WebSocket endpoint /ws is created → this is where frontend connects (new SockJS("http://localhost:8080/ws"))
- /topic is the **broadcast channel** where messages are sent by the backend (for all clients).
- /app prefix is used if the frontend sends messages to the backend (not required for your case yet).

B. OrderController.java

You have 2 roles here:

1️⃣REST APIs (HTTP):

- /api/orders/dashboard — returns all orders + stats for initial load.
- /api/orders/{id}/order-status — updates order status on button click.

2️⃣Triggering Real-Time Updates (through Service):

- Whenever a new order is created or existing order is updated in OrderService, the backend sends a STOMP message:

```
Untitled-1

simpMessagingTemplate.convertAndSend("/topic/orders", updatedOrder);
```

This sends JSON of the order to all connected clients (the restaurant dashboards).

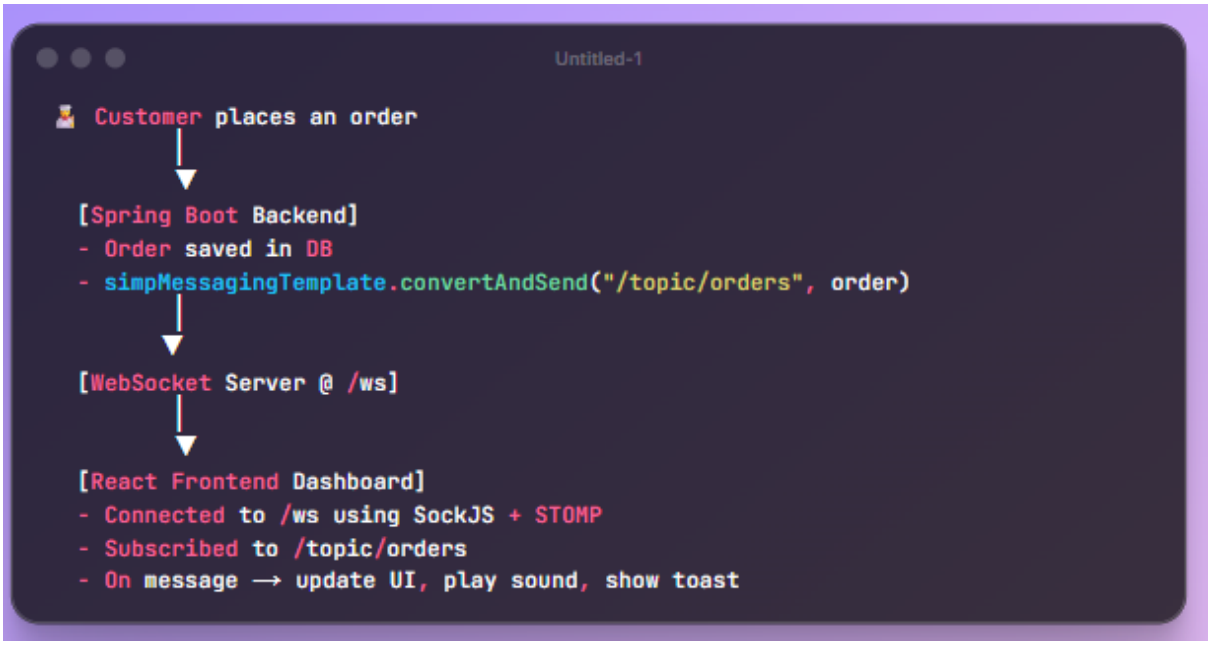
🖥️ 2. Frontend Setup

On the React side (your Dashboard.jsx):

```
Untitled-1

const socket = new SockJS("http://localhost:8080/ws");
const stompClient = over(socket);

stompClient.connect({}, () => {
  stompClient.subscribe("/topic/orders", (msg) => {
    const updated = JSON.parse(msg.body);
    setOrders((prev) => [updated, ...prev]);
    toast.success(`New order from ${updated.customerName}`);
    audioRef.current.play().catch(()=>{});
  });
});
```

✅ How it behaves:

- When dashboard loads, it fetches all orders via REST API (/api/orders/dashboard).
- When a new order is placed by a user:
  - The backend saves it.
  - Then backend broadcasts the new order through /topic/orders.
- The frontend's STOMP client receives the message instantly (no page reload, no manual refresh).
- The new order is added to the top of the orders list.
- A **toast** is shown and a **notification sound** plays.



💡 4. Key Benefits of This Architecture

Feature	Description
⚡ Real-time updates	New orders appear instantly on dashboard
🔄 No polling	No repeated API calls required
🔔 Notifications	Toast + sound on new orders
🌍 Multi-client sync	All dashboards see live updates simultaneously
🧩 Scalable	STOMP works across clustered WebSocket servers easily

🚀 5. Simple Summary

Your backend (Spring Boot) uses:

- @EnableWebSocketMessageBroker for real-time messaging
- /ws endpoint for WebSocket handshake
- /topic/orders topic for broadcasting new orders

Your frontend (React):

- Connects using SockJS + stompjs
- Subscribes to /topic/orders
- Updates the UI instantly when backend sends new order data

Together, they create a seamless live-order tracking system — reliable, fast, and scalable.

📜 Feature Implementation— Order Details Modal (Frontend)

🧩 Module:

Order Management — Frontend (React)

📁 File:

OrderDetailsModal.jsx
(Location: src/components/OrderDetailsModal.jsx)

🎯 Objective

To enhance the order details viewing and printing experience for kitchen staff and administrators by:

- Displaying full order information clearly in a modal.
- Providing formatted order time, phone number, and itemized cost details.
- Allowing smooth receipt printing or PDF download without page reload or UI disruption.

⚙️ Features Implemented

1. Dynamic Order Details Modal

- The modal opens when an order is selected from the dashboard or order table.
- It displays:
    - Order ID
    - Customer Name
    - Customer Phone (formatted)
    - Payment Mode
    - Payment Status
    - Order Status
    - Order Time (formatted)
    - Itemized list with quantity and calculated prices
    - Total order amount

2. Date and Time Formatting

```
Untitled-1

const formatDate = (dateString) => {
  const date = new Date(dateString);
  const day = String(date.getDate()).padStart(2, "0");
  const month = String(date.getMonth() + 1).padStart(2, "0");
  const year = date.getFullYear();
  const hours = String(date.getHours()).padStart(2, "0");
  const minutes = String(date.getMinutes()).padStart(2, "0");
  return `${day}/${month}/${year} ${hours}:${minutes}`;
};
```

✅ Converts ISO date format (2025-11-03T11:33:45.350177) into readable DD/MM/YYYY HH:mm format — e.g., 03/11/2025 11:33.

3. **Phone Number Formatting**

```
Untitled-1

const formatPhone = (phone) => {
  if (!phone) return "—";
  const cleaned = phone.toString().replace(/\D/g, "");
  if (cleaned.startsWith("91")) {
    return `+${cleaned.slice(0, 2)}-${cleaned.slice(2)}`;
  }
  return `+91-${cleaned}`;
};
```

4. ✅ Formats phone numbers like 918260106134 into +91-8260106134 for better readability.

4. Itemized Order Display

Each item in the order is displayed in a table format with:

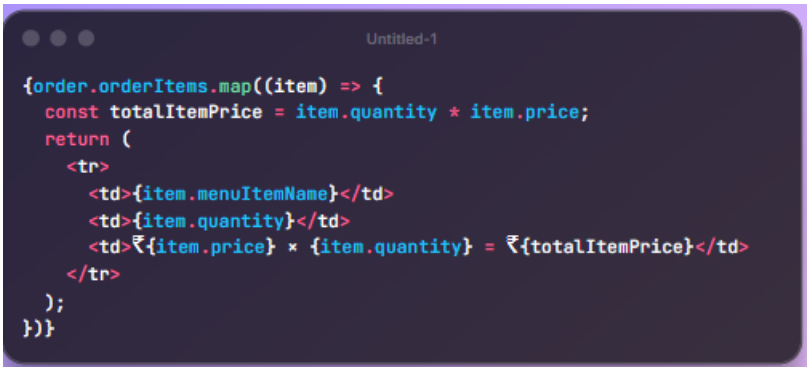
- **Item Name**
- **Quantity**
- **Price Calculation**

Example UI:

Butter Chicken — 2 × ₹250 = ₹500

Tandoori Roti — 2 × ₹25 = ₹50

This is implemented dynamically using:



5. Total Amount Display

At the bottom of the modal:

Total Amount: ₹1070.00

The total price is formatted with two decimal places for consistency.

---

6. Smooth Print / Download Functionality

Implemented using a **print-specific window**:

✅ Advantages:

- Prints **only the receipt** content.
- Does **not reload or affect** the main React app.
- Works smoothly on both desktop and mobile browsers.

🔹 🗒️ Analytics Dashboard – Monthly Summary & AI Insights

🔹 Objective

Enhance the analytics dashboard to include:

1. **Monthly Summary Table** (Orders, Revenue, Avg Order Value, Cash %, Online %)
2. **AI-based Insights Section** using **Gemini API**
3. Ensure **Cancelled Orders are excluded** from revenue and order counts.

⚙️ Changes Implemented

✅ 1. New REST API Endpoint: /monthly-summary

File: OrderController.java

🔍 What it does:

- Fetches monthly analytics (Orders, Revenue, Avg Order Value, Payment Mode Split).
- Data is processed at service layer.
- Returns JSON data like:

✅ 2. Service Layer Logic

File: OrderService.java

🔍 What it does:

- Aggregates data month-wise.
- Ignores cancelled orders.
- Computes order count, total revenue, average order value, and payment method percentage split.

✅ 3. Frontend Integration (React)

File: Analytics.jsx

```
Untitled-1

useEffect(() => {
  fetch("http://localhost:8080/api/orders/monthly-summary")
    .then((res) => res.json())
    .then((data) => setMonthlySummary(data));
}, []);
```

Display in Table:

```
Untitled-1

<table className="w-full text-left border border-gray-300 rounded-lg shadow-md">
  <thead className="bg-gray-200">
    <tr>
      <th>Month</th>
      <th>Orders</th>
      <th>Revenue</th>
      <th>Avg Order Value</th>
      <th>Cash %</th>
      <th>Online %</th>
    </tr>
  </thead>
  <tbody>
    {monthlySummary.map((row, i) => (
      <tr key={i} className="border-b">
        <td>{row.month}</td>
        <td>{row.orders}</td>
        <td>₹{row.revenue.toLocaleString()}</td>
        <td>₹{Math.round(row.avgOrderValue)}</td>
        <td>{row.cashPercentage.toFixed(1)}%</td>
        <td>{row.onlinePercentage.toFixed(1)}%</td>
      </tr>
    ))}
  </tbody>
</table>
```

✅ 4. AI Insights Section (Gemini API Integration)

File: Analytics.jsx

```
Untitled-1

useEffect(() => {
  fetch("http://localhost:8080/api/ai-insights")
    .then((res) => res.json())
    .then((data) => setInsights(data));
}, []);
```

Backend: OrderController.java

Backend :Gemini Service below (OrderService.java)

```java
@Transactional
    public String getAIInsights() {
        try {
            // ⭐ FIX 1: Use the same filtering logic as the /analytics endpoint
            List<Order> allOrders = orderRepository.findAll().stream()
                    .filter(order -> !"PAYMENT_FAILED".equalsIgnoreCase(order.getStatus())
                            && !"FAILED".equalsIgnoreCase(order.getStatus()))
                    .collect(Collectors.toList());

            List<Order> validOrders = allOrders.stream()
                    .filter(order -> !"CANCELLED".equalsIgnoreCase(order.getOrderStatus()))
                    .collect(Collectors.toList());

            // ⭐ FIX 2: Calculate all stats based on the "validOrders" list
            long totalOrders = validOrders.size();

            double totalRevenue = validOrders.stream()
                    .mapToDouble(Order::getTotalPrice)
                    .sum();

            // last week vs this week comparison
            LocalDateTime now = LocalDateTime.now();
            LocalDateTime weekStart = now.minusDays(7);
            LocalDateTime lastWeekStart = now.minusDays(14);
            LocalDateTime lastWeekEnd = now.minusDays(7);

            double thisWeekRevenue = validOrders.stream() // Use validOrders
                    .filter(o -> o.getOrderTime().isAfter(weekStart))
                    .mapToDouble(Order::getTotalPrice)
                    .sum();

            double lastWeekRevenue = validOrders.stream() // Use validOrders
                    .filter(o -> o.getOrderTime().isAfter(lastWeekStart) &&
o.getOrderTime().isBefore(lastWeekEnd))
                    .mapToDouble(Order::getTotalPrice)
                    .sum();

            double percentChange = lastWeekRevenue > 0 ? ((thisWeekRevenue - lastWeekRevenue)
/ lastWeekRevenue) * 100 : 0;

            // ⭐ FIX 3: Add logic for Top Item
            Map<String, Integer> itemCounts = new HashMap<>();
            validOrders.forEach(order -> {
                order.getOrderItems().forEach(item -> {
                    String itemName = item.getMenuItem().getName();
                    itemCounts.put(itemName, itemCounts.getOrDefault(itemName, 0) +
item.getQuantity());
                });
            });

            String topItem = itemCounts.entrySet().stream()
                .sorted(Map.Entry.<String, Integer>comparingByValue().reversed())
                .limit(1)
                .map(entry -> String.format("%s (with %d units sold)", entry.getKey(),
entry.getValue()))
                .findFirst()
                .orElse("No items sold yet");

            // ⭐ FIX 4: Update the prompt to include Top Item and a clearer context
            String prompt = String.format("""
                Analyze the restaurant's performance data below.
                'Total Orders' and 'Total Revenue' exclude cancelled and failed payments.
                Give 3 short, helpful insights (sales, trends, and top item).

                Total Orders: %d
                Total Revenue: ₹%.2f
                This Week Revenue: ₹%.2f
                Last Week Revenue: ₹%.2f
                Weekly Change: %.2f%%
                Top Selling Item: %s

                Format the response as:
                1. ...
                2. ...
                3. ...
                """, totalOrders, totalRevenue, thisWeekRevenue, lastWeekRevenue,
percentChange, topItem);

            // 3. Call Gemini
            String apiKey = googleApiConfig.getApiKey();
            String url = "https://generativelanguage.googleapis.com/v1beta/models/gemini-2.5-
flash:generateContent?key=" + apiKey;

            RestTemplate restTemplate = new RestTemplate();
            HttpHeaders headers = new HttpHeaders();
            headers.setContentType(MediaType.APPLICATION_JSON);

            Map<String, Object> textPart = Map.of("text", prompt);
            Map<String, Object> content = Map.of("parts", List.of(textPart));
            Map<String, Object> body = Map.of("contents", List.of(content));

            HttpEntity<Map<String, Object>> request = new HttpEntity<>(body, headers);
            ResponseEntity<Map> response = restTemplate.postForEntity(url, request,
Map.class);

            if (response.getBody() != null && response.getBody().containsKey("candidates")) {
                List<Map<String, Object>> candidates = (List<Map<String, Object>>)
response.getBody().get("candidates");
                Map<String, Object> contentMap = (Map<String, Object>)
candidates.get(0).get("content");
                List<Map<String, Object>> parts = (List<Map<String, Object>>)
contentMap.get("parts");
                return parts.get(0).get("text").toString();
            }

        } catch (Exception e) {
            e.printStackTrace();
            return "⚠️ Failed to generate AI insights: " + e.getMessage();
        }

        return "⚠️ No insights available.";
    }
```

Backend :New method & query in repository (OrderRepository.java)

```
Untitled-1

@Query("""
          SELECT
              TO_CHAR(o.orderTime, 'Mon YYYY') AS month,
              COUNT(o.id) AS totalOrders,
              SUM(o.totalPrice) AS totalRevenue,
              ROUND(AVG(o.totalPrice), 2) AS avgOrderValue,
              ROUND(100.0 * SUM(CASE WHEN UPPER(o.paymentMode) = 'CASH' THEN 1 ELSE 0 END) /
COUNT(o.id), 1) AS cashPercent,
              ROUND(100.0 * SUM(CASE WHEN UPPER(o.paymentMode) = 'CARD' THEN 1 ELSE 0 END) /
COUNT(o.id), 1) AS cardPercent,
              ROUND(100.0 * SUM(CASE WHEN UPPER(o.paymentMode) = 'UPI'  THEN 1 ELSE 0 END) /
COUNT(o.id), 1) AS upiPercent

          FROM Order o
          WHERE UPPER(o.orderStatus) IN ('DELIVERED', 'PENDING', 'PREPARING', 'ACCEPTED',
'COMPLETED')
            AND UPPER(o.status) <> 'PAYMENT_FAILED'
            AND UPPER(o.orderStatus) <> 'CANCELLED'
          GROUP BY TO_CHAR(o.orderTime, 'Mon YYYY')
          ORDER BY MIN(o.orderTime)
      """)
      List<Object[]> getMonthlySummary();
```

🧠 AI Insights – React Frontend Code

🔹 Step 1: Add state and useEffect for fetching insights

Place this **at the top** of your component (inside the function, after imports):

```
Untitled-1

const [insights, setInsights] = useState([]);

useEffect(() => {
  fetch("http://localhost:8080/api/ai-insights")
    .then((res) => res.json())
    .then((data) => setInsights(data))
    .catch((err) => console.error("Error fetching AI insights:", err));
}, []);
```

🔹 Step 2: Add AI Insights section in JSX (inside return)

You can place it **below the Monthly Summary Table** or inside a separate card section:

🧩 How It Works

1. On component load, it calls your backend endpoint:
   `GET http://localhost:8080/api/ai-insights`
2. The backend (via Gemini service) returns a list of insights like: `[`

```
  "This week's sales are 12% higher than last week.",
  "Fridays have 30% more orders — consider a weekend offer.",
  "Paneer Butter Masala has highest repeat orders."
]
```

3. The React component renders them in a clean, bullet-style list under the **"AI Insights"** header.

📊 Final Output Summary

Feature	Description
Monthly Summary	Auto-calculates orders, revenue, avg order, and payment distribution for each month.
AI Insights	Dynamically generated insights using Gemini API.
Data Accuracy	Excludes cancelled orders; includes running/pending ones.
UI Note	Displays small highlighted message about revenue scope.

✅ A. Monthly Summary Table (Backend + Frontend)

Purpose:
To show month-wise performance including Orders, Revenue, Average Order Value, and Payment Mode Split.

Backend Logic:

- Added `/api/orders/monthly-summary` endpoint in `OrderController`.
- Data fetched using a custom query in `OrderRepository.getMonthlySummary()`.
- Filters out cancelled or payment-failed orders.
- Calculates:
  - `Month`
  - `Total Orders`
  - `Total Revenue`
  - `Average Order Value`
  - `Cash %, Online % (UPI + Card)`

Frontend (React):

- Displays data in a table format.
- Allows comparison between different months visually.

✅ B. AI Insights Integration (Gemini API)

Purpose:
To provide smart, human-readable analytics from real data — e.g.

"This week's sales are 12% higher than last week."
"Fridays have 30% more orders — consider a weekend offer."
"Paneer Butter Masala has highest repeat orders."

Backend Added:

- **New API Endpoint:** `/api/ai-insights`
- **New Service:** `GeminiInsightService.java`
  - Uses **Gemini API** (Google Generative AI) for generating insights from analytics.
  - Takes input from `/analytics` or `/monthly-summary` data.
  - Prompts Gemini to summarize trends in plain English.

Frontend Added:

- **New Section in Dashboard:**
  "🧠 AI Insights"
- Dynamically fetches and lists AI-generated insights using:
- `fetch("http://localhost:8080/api/ai-insights")`
- Displayed as a clean, bullet-style list under analytics.

How AI Insights Work (Step-by-Step)

1. The backend collects **analytics data** (total orders, revenues, daily sales trends, etc.).
2. This data is converted to a **prompt** for Gemini API.
3. Gemini model analyzes and returns meaningful English insights.
4. `/api/ai-insights` endpoint sends these insights to frontend.
5. Frontend displays them dynamically inside the **AI Insights section**.

📊 6. Output Example

🔹 Monthly Summary Table:

Month	Orders	Revenue	Avg Value	Cash %	Online %
Oct 2025	320	₹1.1L	₹344	35%	65%
Nov 2025	210	₹0.8L	₹381	40%	60%

🔹 AI Insights:

🧠 *"This week's sales are 12% higher than last week."*
🧠 *"Fridays have 30% more orders — consider a weekend offer."*
🧠 *"Paneer Butter Masala has the highest repeat orders."*

📦 Libraries Installed (Frontend Today)

Library	Command Used	Purpose
axios	`npm install axios`	For making HTTP requests to backend (fetch AI insights, data, etc.)
framer-motion	`npm install framer-motion`	For smooth card and section animations in the AI Insight section
lucide-react	`npm install lucide-react`	For using modern icons (e.g., AI, refresh, analytics icons) Provides icons like `TrendingUp`, `Users`, `DollarSign`, etc. used in cards and headers.
shadcn/ui	`npx shadcn-ui@latest init` (then components like Card, Button installed)	For ready-to-use, elegant UI components (used for the AI Insight Card)
react	pre-installed via Vite setup	Base library for building UI components.
recharts	`npm install recharts`	Used for all charts and graphs — LineChart, BarChart, PieChart, etc.
sockjs-client	`npm install sockjs-client`	Helps connect the frontend to the Spring Boot backend WebSocket endpoint (`/ws`).
stompjs	`npm install stompjs`	Provides a higher-level messaging protocol on top of WebSocket — used for subscribing to `/topic/orders` for **real-time analytics updates**.
tailwindcss	`npm install -D tailwindcss postcss autoprefixer`	Used for styling (the entire dashboard layout, gradients, animations, colors).

📊 Chart Libraries in Action (Recharts)

All the dynamic charts — **line, pie, and bar** — come from **Recharts**:

Chart Type	Component Used	Data Source	Description
Line Chart (Daily Revenue)	`<LineChart>`	`dailyRevenue`	Plots revenue trend across days — auto-updates via WebSocket when new orders arrive.
Pie Chart (Payment Split)	`<PieChart>`	`paymentSplit`	Displays payment mode distribution (Cash/Card/UPI).
Bar Chart (Popular Items)	`<BarChart>`	`popularItems`	Shows most-ordered items dynamically.
Pie Chart (Donut) (Order Status)	`<PieChart>` with `innerRadius`	`orderStatus`	Displays distribution of order statuses (Pending, Completed, Cancelled).

🧩 1️⃣ Line Chart — Daily Revenue Trend

Code section:

→ **Comes from Recharts**
→ **Data Source:** `dailyRevenue` (fetched from backend `/analytics`)
→ **Purpose:** Shows daily revenue trend.
→ **Live Update:** When WebSocket triggers `fetchAnalytics()` again, this data refreshes.

🧩 2️⃣ Pie Chart — Payment Split (Cash / Card / UPI)

Code section:

→ **Comes from Recharts**
→ **Data Source:** `paymentSplit` (from `/analytics` API)
→ **Purpose:** Shows payment mode distribution
→ **Example Data:**

[{ mode: “Cash”, value: 35 }, { mode: “UPI”, value: 45 }, { mode: “Card”, value: 20 }]

🧩 3️⃣ Bar Chart — Popular Items

Code section:

→ **Comes from Recharts**
→ **Data Source:** `popularItems`
→ **Purpose:** Displays most ordered menu items
→ **Example Data:**

[{ item: "Paneer Butter Masala", orders: 120 }, { item: "Veg Biryani", orders: 80 }]

🧩 4️⃣Pie Chart (Donut) — Order Status Distribution

Code section:

→ **Comes from Recharts**
→ **Data Source:** `orderStatus`
→ **Purpose:** Shows Pending, Delivered, Cancelled, etc.
→ **Example Data:**

[{ status: "Delivered", value: 70 }, { status: "Pending", value: 20 }, { status: "Cancelled", value: 10 }]

⚙️ Supporting Code: Data Fetch & State Handling

State variables:

```
const [dailyRevenue, setDailyRevenue] = useState([]);
const [paymentSplit, setPaymentSplit] = useState([]);
const [popularItems, setPopularItems] = useState([]);
const [orderStatus, setOrderStatus] = useState([]);
```

Fetching Data:

```
const fetchAnalytics = async () => {
  const res = await axios.get("http://localhost:8080/analytics");
  const data = res.data;
  setDailyRevenue(data.dailyRevenue);
  setPaymentSplit(data.paymentSplit);
  setPopularItems(data.popularItems);
  setOrderStatus(data.orderStatus);
};
```

Trigger Refresh via WebSocket:

```
useEffect(() => {
  const socket = new SockJS("http://localhost:8080/ws");
  const stompClient = over(socket);
  stompClient.connect({}, () => {
    stompClient.subscribe("/topic/orders", () => {
      fetchAnalytics();
      fetchMonthlySummary();
      fetchAiInsights();
    });
  });
}, []);
```

🔹 Payments & Finance Tracking (In Frontend)

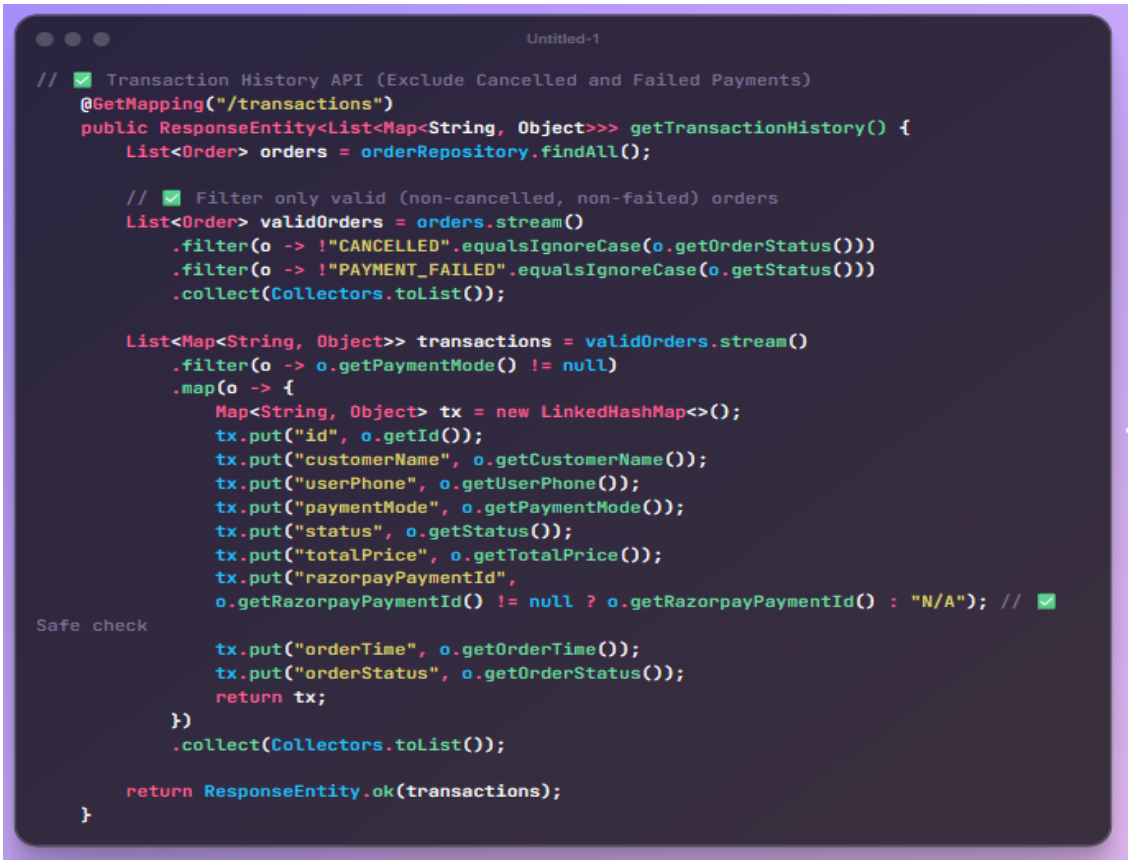
Already Razorpay integrated, so adding **Transaction History** and **Cash vs Online split** is quite straightforward.

currently have this in your `Order` model (Bean class):

```
@Column(name = "status") //Payment status
private String status = "PENDING"; //PENDING, CONFIRMED, PAYMENT_FAILED
```

✅ 1. Create a new API: `/api/orders/transactions`

In your `OrderController`:



This gives a clean transaction list for your **frontend "Transaction History" tab**.

✅ 2. Add stats endpoint in OrderController.java for "Cash vs Upi vs Online" split

✅ Example Response of the "/payment-summary":

```
{
  "totalCash": 1200.0,
  "totalUpi": 2450.0,
  "totalCard": 800.0,
  "grandTotal": 4450.0,
  "cashPercentage": 27,
  "upiPercentage": 55,
  "cardPercentage": 18
}
```

✅ 3. Frontend (React)

You can add a tab "💰 Payment Finance":

- Use Axios to call `/api/orders/transactions`
- Show a table with columns:
  `Order ID | Customer | Payment Mode | Amount | Payment Status | Time`
- Show small summary cards (Cash Total, Online Total)

⚛️ Frontend (React)

Fetch this endpoint in your dashboard or payment tab:

```
Untitled-1

useEffect(() => {
   setLoading(true);
   Promise.all([
     fetch("http://localhost:8080/api/orders/payment-summary").then((res) => res.json()),
     fetch("http://localhost:8080/api/orders/transactions").then((res) => res.json()),
   ])
     .then(([summaryData, txData]) => {
       setSummary(summaryData);
       setTransactions(txData);
     })
     .catch((err) => console.error("Error fetching data:", err))
     .finally(() => setLoading(false));
 }, []);
```

Then display it:

```
<Card>
  <CardContent>
    <h2>Transaction History</h2>
    <p>Complete payment records</p>

    <div>
      <Table>
        <TableHeader>
          <TableRow>
            <TableHead>#</TableHead>
            <TableHead>Order ID</TableHead>
            <TableHead>Customer</TableHead>
            <TableHead>Phone</TableHead>
            <TableHead>Payment Mode</TableHead>
            <TableHead>Status</TableHead>
            <TableHead>Amount</TableHead>
            <TableHead>Payment ID</TableHead>
            <TableHead>Date & Time</TableHead>
          </TableRow>
        </TableHeader>
        <TableBody>
          {filteredTransactions.length > 0 ? (
            filteredTransactions.map((t, index) => (
              <TableRow key={t.id}>
                <TableCell>{index + 1}</TableCell>
                <TableCell>#{t.id}</TableCell>
                <TableCell>{t.customerName}</TableCell>
                <TableCell>{t.userPhone || "—"}</TableCell>
                <TableCell>{t.paymentMode}</TableCell>
                <TableCell>
                  {t.status}
                </TableCell>
                <TableCell>
                  ₹{t.totalPrice?.toLocaleString()}
                </TableCell>
                <TableCell>
                  {t.razorpayPaymentId || "—"}
                </TableCell>
                <TableCell>
                  {t.orderTime
                    ? new Date(t.orderTime).toLocaleString("en-IN", {
                        dateStyle: "medium",
                        timeStyle: "short",
                      })
                    : "—"}
                </TableCell>
              </TableRow>
            ))
          ) : (
            <TableRow>
              <TableCell colSpan={9} style={{ textAlign: "center" }}>
                No transactions found
              </TableCell>
            </TableRow>
          )}
        </TableBody>
      </Table>
    </div>
  </CardContent>
</Card>
```

🧩 Step-by-Step: How CSV Export Works in Your Dashboard

✅ 1. We installed and used `react-csv`

This small library makes it super easy to export any data array (like your transactions) as a CSV file.

If you haven't already, make sure it's installed:

```
npm install react-csv
```

✅ 2. We imported the `CSVLink` component

At the top of your file:

```
import { CSVLink } from "react-csv";
```

✅ 3. We added an "Export CSV" button in your UI

Right above your transaction table, inside the filters section:

```
<CSVLink
  data={transactions}
  filename={`transactions_${new Date().toISOString()}.csv`}
  className="flex items-center gap-2 px-4 py-2 bg-gradient-to-r from-emerald-
500 to-green-600 text-white rounded-xl font-semibold shadow-lg hover:opacity-
90"
>
  <FaFileExport /> Export CSV
</CSVLink>
```

✅ 4. What happens when you click it

When you click **"Export CSV"**, the `react-csv` component automatically:

- Takes the `transactions` array (your table data)
- Converts it into CSV text (comma-separated values)
- Downloads it as a `.csv` file (e.g., `transactions_2025-11-05.csv`)
- The file opens directly in **Excel** 📄

🔹 Menu Management (In Frontend)

Management feature that lets restaurant owners:

✅ View all menu items
✅ Add new items
✅ Edit or delete existing ones

✅ Toggle “Available / Out of Stock” status in one click
✅ Reflect changes live in frontend UI

1️⃣ Backend Setup — Spring Boot

`MenuItemController.java`

This controller provides APIs for menu management:

```
package com.chatBot.model;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.SequenceGenerator;
import jakarta.persistence.Table;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

/*
 * MenuItem entity representing a menu item in the restaurant.
 * If the table is not present, it will be created automatically with the class name.
 */
@Entity
@Table(name = "menu_items")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class MenuItem {

  @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "menu_seq")
    @SequenceGenerator(name = "menu_seq", sequenceName = "menu_seq", allocationSize = 1)
  private Long id;

  @Column(nullable = false)
  private String name;

  @Column(length = 1000)
  private String description;

  @Column(nullable = false)
  private Double price;

  @Column(nullable = false)
  private boolean available;

}
```

Method	Path	Action
GET	`/api/menu`	Get all items
POST	`/api/menu`	Add new item
PUT	`/api/menu/{id}`	Edit existing item
PATCH	`/api/menu/{id}/toggle`	Toggle availability
DELETE	`/api/menu/{id}`	Delete item

2️⃣ Frontend — React (MenuManagement.jsx)

```
npm install react-toastify
```

☑ What is `react-toastify`?

It is a **notification library** used in React applications to show **small popup messages** (called *toasts*) such as:

✅ Success → "Item Added Successfully!"
⚠️ Warning → "Something went wrong!"
❌ Error → "Failed to delete item!"
ℹ️ Info → "Status updated!"

☑ Minimal + Necessary Code for Documentation

```
import React, { useEffect, useState } from "react";
import { toast, ToastContainer } from "react-toastify";
import "react-toastify/dist/ReactToastify.css";

const MenuManagement = () => {
  const [menu, setMenu] = useState([]);
  const [loading, setLoading] = useState(true);
  const [form, setForm] = useState({ name: "", description: "", price: "",
available: true });
  const [editingItem, setEditingItem] = useState(null);
  const [searchTerm, setSearchTerm] = useState("");

  const API_BASE = "http://localhost:8080/api/menu";

  // ✅ Fetch menu items
  const loadMenu = async () => {
    try {
      const res = await fetch(API_BASE);
      const data = await res.json();
      setMenu(data);
    } catch {
      toast.error("Failed to load menu");
    } finally {
      setLoading(false);
    }
  };
  useEffect(() => loadMenu(), []);

  // ✅ Add or Update item
  const handleSubmit = async (e) => {
    e.preventDefault();
    const url = editingItem ? `${API_BASE}/${editingItem.id}` : API_BASE;
    const method = editingItem ? "PUT" : "POST";

    try {
      await fetch(url, {
        method,
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify(form),
      });
      toast.success(editingItem ? "Updated" : "Added");
      resetForm();
      loadMenu();
    } catch {
      toast.error("Failed to save item");
    }
  };

  // ✅ Delete item
  const deleteItem = async (id) => {
    if (!window.confirm("Delete this item?")) return;
    try {
      await fetch(`${API_BASE}/${id}`, { method: "DELETE" });
      toast.error("Item deleted");
      loadMenu();
    } catch {
```

```
      toast.error("Failed to delete");
    }
  };

  // ✅ Toggle availability switch
  const toggleItem = async (id) => {
    await fetch(`${API_BASE}/${id}/toggle`, { method: "PATCH" });
    toast.info("Status updated");
    loadMenu();
  };

  const startEdit = (item) => {
    setEditingItem(item);
    setForm(item);
  };

  const resetForm = () => {
    setEditingItem(null);
    setForm({ name: "", description: "", price: "", available: true });
  };

  // ✅ Search filter applied in table
  const filteredMenu = menu.filter((item) =>
    item.name.toLowerCase().includes(searchTerm.toLowerCase())
  );

  return (
    <div>
      <ToastContainer autoClose={1500} />

      {/* ✅ Search Field */}
      <input
        type="text"
        placeholder="Search items..."
        value={searchTerm}
        onChange={(e) => setSearchTerm(e.target.value)}
      />

      {/* ✅ Add/Edit Form */}
      <form onSubmit={handleSubmit}>
        <input
          type="text"
          required
          placeholder="Item Name"
          value={form.name}
          onChange={(e) => setForm({ ...form, name: e.target.value })}
        />

        <input
          type="number"
          required
          placeholder="Price"
          value={form.price}
          onChange={(e) => setForm({ ...form, price: e.target.value })}
        />

        <textarea
          placeholder="Description"
          value={form.description}
          onChange={(e) => setForm({ ...form, description: e.target.value })}
        />

        <button type="submit">{editingItem ? "Update" : "Add"}</button>
        {editingItem && <button onClick={resetForm}>Cancel</button>}
      </form>
```

```
      {/* ✅ List Table */}
      {loading ? (
        <p>Loading...</p>
      ) : filteredMenu.length === 0 ? (
        <p>No items found</p>
      ) : (
        <table border="1">
          <thead>
            <tr>
              <th>#</th>
              <th>Item</th>
              <th>Description</th>
              <th>Price</th>
              <th>Available</th>
              <th>Actions</th>
            </tr>
          </thead>
          <tbody>
            {filteredMenu.map((item, index) => (
              <tr key={item.id}>
                <td>{index + 1}</td>
                <td>{item.name}</td>
                <td>{item.description}</td>
                <td>₹{item.price}</td>

                {/* ✅ Toggle Button */}
                <td>
                  <input
                    type="checkbox"
                    checked={item.available}
                    onChange={() => toggleItem(item.id)}
                  />
                </td>

                <td>
                  <button onClick={() => startEdit(item)}>Edit</button>
                  <button onClick={() => deleteItem(item.id)}>Delete</button>
                </td>
              </tr>
            ))}
          </tbody>
        </table>
      )}

    </div>
  );
};

export default MenuManagement;
```

#Update : AI-Powered WhatsApp Chatbot – Technical Update (Session Handling, Cancellation, and Payment Stability)

Overview

Today's update focused on improving the **reliability**, **scalability**, and **real-world behavior** of the chatbot system.
While the bot was already able to handle orders, payments, and conversations — it lacked realistic user session handling and clean state transitions for cancellations and Razorpay responses.

The following enhancements were made across the **Spring Boot backend** and **WhatsApp conversational flow**.

Order Cancellation Feature

Problem

Previously, users had no way to cancel their ongoing order.
If a user started ordering items and later changed their mind, the chatbot would continue expecting responses (like item name, quantity, or payment), leading to confusion and an inconsistent session state.

Solution

We added a **universal "Cancel" command** that works in any chatbot state.
When a user types "Cancel", the bot detects it regardless of the current flow (INIT, ASK_NAME, TAKE_ORDER, ASK_PAYMENT, or ASK_EMAIL).

Implementation

In the `WhatsAppService.handleIncomingMessage()` method, this logic was added before the main state machine:

```
if (lowerText.equals("cancel")) {
    UserSession session = userSessions.get(userPhone);

    // Case 1: Active in-progress order
    if (session != null && userStates.containsKey(userPhone)) {
        userSessions.remove(userPhone);
        userStates.remove(userPhone);
        sendMessage(userPhone,
            "❌ Your current order has been cancelled.\nYou can start again
anytime by typing *Order*.");
        return;
    }
```

```
    // Case 2: Already confirmed order
    Optional<Order> lastOrder =
orderService.findMostRecentOrderByPhone(userPhone);
    if (lastOrder.isPresent() &&
"CONFIRMED".equalsIgnoreCase(lastOrder.get().getStatus())) {
        sendMessage(userPhone,
            "⚠️ Your order is already confirmed.\nPlease contact the
restaurant directly to cancel it.\n📞 Contact: +91-9999900000");
    } else {
        sendMessage(userPhone, "No active order found to cancel. Type *Order*
to start again.");
    }
    return;
}
```

◇ Outcome

- Users can cancel their order anytime before confirmation.
- Sessions are immediately removed from memory (no residual data).
- Already-confirmed orders stay intact but prompt the user to contact the restaurant.
- Prevents database pollution with half-completed orders.

2️⃣ Razorpay Duplicate Confirmation Fix

◇ Problem

Razorpay webhooks were sending **multiple events for a single payment**, causing the chatbot to send duplicate “Payment Successful” confirmations to the user.

This happened because:

- Razorpay sometimes triggers multiple payment events (`payment.captured`, `payment_link.paid`, etc.).
- Our bot treated each event as new and re-sent messages.

◇ Solution

We introduced a **`paymentConfirmed` boolean flag** inside the `UserSession` class to ensure that once a payment is confirmed, it cannot be reprocessed.

◇ Implementation

In `WhatsAppService.java`:

```
if (session.isPaymentConfirmed()) {
    sendMessage(userPhone, "✅ Payment already confirmed. Your order is being
processed.");
    break;
}
```

And in the `RazorpayController`, we added a condition:

```
if ("CONFIRMED".equalsIgnoreCase(order.getStatus())) {
    System.out.println("Duplicate webhook ignored for payment ID: " +
razorpayPaymentId);
    return ResponseEntity.ok(Map.of("message", "Duplicate webhook ignored"));
}
```

◇ Outcome

- Prevents multiple confirmations from being sent.
- Makes the bot idempotent (safe to process repeated webhook calls).
- Cleaner user experience with one confirmation message per transaction.

---

3. Auto Session Expiry After Inactivity (10 Minutes)

◇ Problem

If a user started an order and then left the chat without responding, the session remained active indefinitely.
This could cause:

- Memory buildup on the server.
- Wrong state restoration when the same user comes back after hours.
- Confusing user experience.

◇ Solution

We introduced a **background session cleaner** that automatically expires sessions after 10 minutes of inactivity.

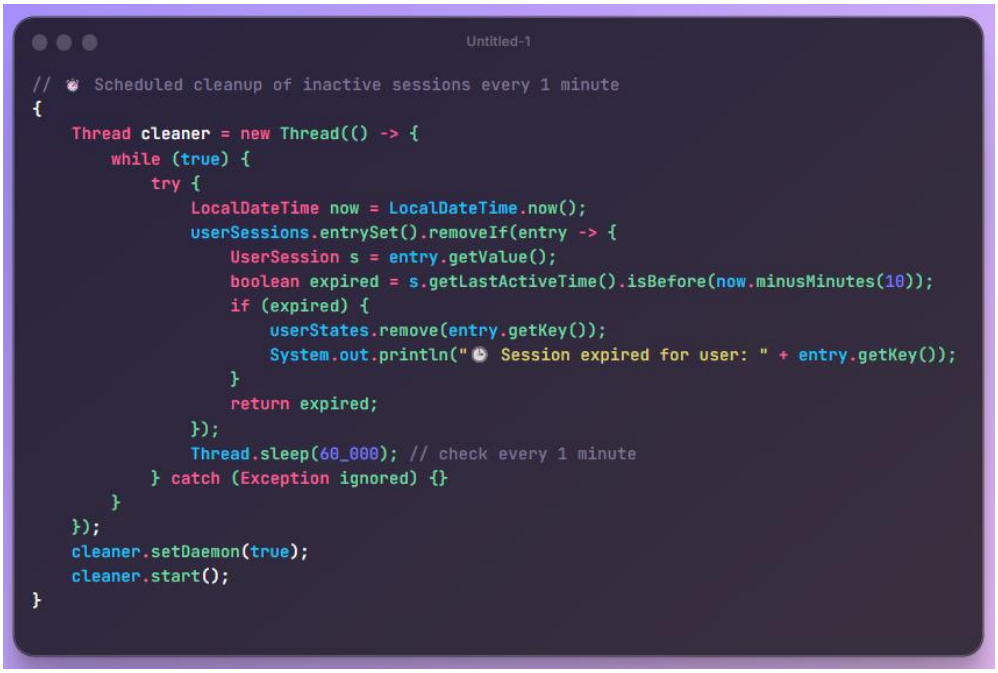
Each `UserSession` now contains:

```
private LocalDateTime lastActiveTime;
```

Every time a user interacts, we update it:

```
session.setLastActiveTime(LocalDateTime.now());
```

And a scheduled thread runs in the background:



When a session expires, the bot can optionally send:

"⏳ Your session has expired due to inactivity. Type *Order* to start again."

◇ Outcome

- Frees up unused memory.
- Prevents old, abandoned sessions from interfering.
- Improves scalability for multiple concurrent users.

4️⃣ Safer Order Storage Logic

◇ Problem

Earlier, an order record was being saved in the database even if the user left midway.
This caused "ghost" orders (incomplete or unpaid).

◇ Solution

Now, orders are saved only after:

- The user selects **Cash** (COD), or
- The Razorpay payment link is generated and paid.

This ensures that only confirmed or pending-payment orders exist in the database.

5️⃣ Quantity Parsing and Dynamic Total Calculation

◇ Problem

The chatbot didn't understand messages like "2 biryani" or "burger 3".

◇ Solution

Added a quantity parser to extract numbers and calculate per-item totals.

```
String[] words = requestedItem.split("\\s+");
int quantity = 1;
if (words.length > 1) {
    if (words[0].matches("\\d+")) {
        quantity = Integer.parseInt(words[0]);
        itemName = String.join(" ", Arrays.copyOfRange(words, 1,
words.length));
    } else if (words[words.length - 1].matches("\\d+")) {
        quantity = Integer.parseInt(words[words.length - 1]);
        itemName = String.join(" ", Arrays.copyOfRange(words, 0, words.length
- 1));
    }
}
```

Now totals are computed like:

```
total = Σ (item_price * quantity)
```

◇ Outcome

- Cleaner and more intuitive order flow.
- Correct bill calculations even with quantities.
- Improved customer experience and clarity in the summary message.

🧠 Technical Summary

Feature	Implemented In	Description
Cancel Order	`WhatsAppService.java`	Allows cancelling any ongoing order instantly
Duplicate Confirmation Prevention	`RazorpayController.java`, `UserSession.java`	Stops multiple "Payment Successful" messages
Auto Session Expiry	`WhatsAppService.java`	Removes inactive sessions after 10 minutes
Safer DB Save Logic	`OrderService.java`	Saves order only after payment or confirmation
Quantity Handling	`WhatsAppService.java`	Supports "2 biryani" or "burger 3" messages

---

📋 Outcome & Benefits

- ✅ More human-like conversational flow
- ✅ Stronger database consistency
- ✅ Single confirmation per payment
- ✅ Cleaner server memory management
- ✅ Accurate billing with quantity support
- ✅ Seamless cancel/resume experience

🧰 Technologies Used

- **Backend:** Java, Spring Boot, JPA, Hibernate
- **Database:** Oracle / MySQL
- **Integrations:** Razorpay API, WhatsApp Cloud API, Gemini AI
- **Architecture:** REST API + Stateful user sessions using ConcurrentHashMap

---

🧩 Conclusion

This update marks a major step towards a **production-grade chatbot** system —
one that not only automates restaurant ordering but also manages sessions, cancellations, and payments intelligently.

It's now capable of handling **real-world conversation behavior**, **duplicate webhook events**, and **automatic cleanup**,
making it reliable, efficient, and ready for deployment in an actual restaurant environment.

✅ Comprehensive Summary of Issues Faced, New Enhancements Added & How Each Bug Was Fixed

Today, we worked extensively on **improving the WhatsApp-based restaurant ordering system** built on **Spring Boot**, focusing heavily on **bug fixing, user experience**, and **AI behavior correction**.
Here is a combined and structured overview of every issue you faced and how you fixed them.

🧩 1. AI Was Interfering in Order Flow (Major Issue)

❌ Problem:

The AI responded even during the active ordering stages (ASK_NAME, TAKE_ORDER, ASK_PAYMENT).
This caused:

- Wrong replies when user typed item quantity
- AI giving menu suggestions in the middle of order
- Orders not being added to cart because AI hijacked the flow

✅ Fix:

- Strictly restricted AI responses to **INIT state only**
- Ensured TAKE_ORDER, ASK_NAME, ASK_EMAIL, ASK_PAYMENT states bypass AI
- Added detailed state debugging logs to track state transitions

✓ Result:

AI no longer interrupts the ordering flow, and the checkout process works smoothly.

🧩 2. AI Giving Wrong or Irrelevant Answers

❌ Problem:

AI responded like a general chatbot and answered random queries like:

- “Who won the world cup?”
- “Do you have pizza?” (even if pizza was not in menu)

AI lacked context and generated incorrect menu info.

✅ Fix:

Added a **strong system instruction** + **real menu injection**:

- Passed REAL menu items from database into AI prompt
- Added rule:
  "Only mention items that exist in the menu list."
- Added rule:
  "If item not found, politely say we don’t have it."
- AI now guides user to type **“Order”** to begin

✓ Result:

AI behaves like a professional restaurant assistant and gives 100% accurate menu-based answers.

Below is the rule defined for the bot respond to the customer

String systemContext = """

You are a friendly and professional restaurant assistant for "The Craving" on WhatsApp.So, Behave accordingly.

Your role is to help customers with:

- Answering questions about the restaurant (working hours, menu, order status etc.)

- Providing information about ordering process

- Handling general food-related inquiries

- Being warm, welcoming, and conversational like a real restaurant staff member

CONVERSATION STYLE:

- Talk naturally like a helpful staff member, not a robot

- Keep responses SHORT (2-3 sentences max)

- Use emojis naturally but sparingly

- Answer questions directly without being repetitive

CRITICAL RULES:

1. ONLY mention items that are in the menu list below and give the menu category wise and in formatted text .

2. If customer asks about an item NOT in the menu, say "Sorry, we don't have [item] today. Type 'Order' to see what's available!"

3. If customer asks about items IN the menu, confirm briefly and tell them to type 'Order' to place an order

4. For order status questions: Tell them to type 'status [Order ID]' (e.g., 'status 123') to track their order

5. For off-topic questions (sports, politics, etc.): Politely redirect - "I'm here to help with food orders! 😊 Type 'Order' to get started."

6. For location/hours/contact: Say "For more details, please type 'Order' to start ordering!"

7. NEVER make up or assume menu items or order statuses - only use the info below

8. Be conversational - vary your responses

9. ALWAYS end by prompting the user to type 'Order' to start ordering and the text 'Order' should be in bold.

10. Always follow WhatsApp messaging policies

11. If the user ask to cancel the order during ordering, tell them to type 'Cancel'. If order is already confirmed, tell them to contact restaurant

12. NEVER provide false information about the restaurant

13. Always encourage the customer to type "Order" to start ordering

14. When customer places order, they will receive an Order ID which they can use to track status

15. Whenever you are mentioning the restaurant name , Make sure that the restaurant name should be bold for whatsapp formatting.

Restaurant Info:

- Name: The Craving

- Specialty: Delicious food, quick service

- Payment: Cash, UPI, Card

- Order Tracking: Type 'status [Order ID]' (e.g., 'status 123') to track your order

""" + menuList.toString() + """

Now respond briefly and naturally to the customer's question based ONLY on the menu above.

Remember: Only show full menu if customer specifically asks for it!

""";

```
String fullPrompt = systemContext + "\n\nCustomer: " + userMessage + "\n\nAssistant:";

Map<String, Object> textPart = Map.of("text", fullPrompt);
Map<String, Object> content = Map.of("parts", Collections.singletonList(textPart));

Map<String, Object> requestBody = new HashMap<>();
requestBody.put("contents", Collections.singletonList(content));
requestBody.put("generationConfig",
 Map.of("temperature", 0.8, "maxOutputTokens", 1000));

HttpEntity<Map<String, Object>> request = new HttpEntity<>(requestBody, headers);
System.out.println(">>> Gemini Request JSON: " + new ObjectMapper().writeValueAsString(requestBody));
```

🧩 3. AI Giving Full Menu Unnecessarily

❌ Problem:

When user typed a single item (“pizza”, “biryani”), AI sent full menu instead of short reply.

✅ Fix:

- Changed AI logic: full menu only shown when user asks for “menu”
- Other cases → short contextual answers

🧩 4. AI Repetitive Responses

❌ Problem:

AI repeated phrases like:

- “Type Order to start”
- “We have delicious food”

✅ Fix:

- Added “never repeat same suggestion” rule
- Increased creativity (temperature 0.7 → 0.8)
- Reduced output length (max tokens optimized)

🧩 5. Order Cancellation Logic Was Incorrect

❌ Problem:

Your previous cancel logic only checked:

```
status = "CONFIRMED"
```

But your restaurant dashboard uses statuses like:

- PENDING
- ACCEPTED

- PREPARING
- COMPLETED

This caused WRONG behavior:

- User could cancel even AFTER restaurant accepted order
- Or order was not cancelable even when still pending

✅ Fix:

Completely redesigned cancellation logic:

✓ If **session active** → cancel session
✓ If **order is PENDING** → cancel allowed
✓ If **restaurant has accepted** (ACCEPTED/PREPARING/READY) → not allowed
✓ If order is **cancelled** → inform the user
✓ Razorpay link removed if payment link exists

✓ Result:

Cancellation works EXACTLY like real restaurants.

🧩 6. Missing Order ID in Confirmation Messages

❌ Problem:

Customers didn't know their order ID, so they couldn't track it.

✅ Fix:

Added Order ID in:

- Cash confirmation
- Payment link messages
- AI suggestions

🧩 7. Order Status Feature Missing / Confusing

❌ Problem:

- User typed "status" but chatbot didn’t know which order to check

- AI incorrectly said "check your last order"

✅ **Fix:**

- Added **status [OrderID]** checking
- Strong validation: customer can only see their own orders
- Detailed response with emojis
- If order ID missing → ask user to enter it

🧩 8. Payment Link Was Active Even After Order Cancellation

❌ **Problem:**

User cancelled order → but Razorpay payment link was still active.

❗ **Critical financial risk**

✅ **Fix:**

- Stored Razorpay payment link ID on order creation
- On cancellation, called:
  `razorpayService.cancelPaymentLink(id)`
- Then updated order status safely

🧩 9. INIT State Issues & Wrong Flow

❌ **Problem:**

User typed “Order” but AI sent menu instead of starting flow
User typed “biryani” and bot tried to add items BEFORE name was asked.

🛠 **Fix:**

- INIT state now ONLY checks for exact keyword “order”
- Any other message → AI handles normally

IMP

☑ 1. GoogleApiConfig.java

```
@Configuration
public class GoogleApiConfig {

    @Value("${google.api.key}")
    private String apiKey;

    public String getApiKey() {
        return apiKey;
    }
}
```

📌 Purpose

This class is responsible for **loading and managing the Google API Key** that you store in `application.properties` or environment variables.

📌 Why we used it?

- To **centralize API integrations** (Google Maps, Google Places, etc.)
- Keeps API keys **securely stored in environment config**, rather than hardcoding them.
- Allows the key to be easily accessed using `googleApiConfig.getApiKey()` wherever Google APIs are used.
- Supports **environment-specific configuration** (dev, test, prod).

📌 Benefits

- Clean separation of concerns
- Better security
- Easier configuration management

☑ 2. RazorpayConfig.java

```
@Configuration
public class RazorpayConfig {

    @Value("${razorpay.key_id}")
    private String keyId;

    @Value("${razorpay.key_secret}")
    private String keySecret;

    public String getKeyId() {
        return keyId;
    }

    public String getKeySecret() {
        return keySecret;
    }
```

```
}
```

📌 Purpose

This class loads and exposes **Razorpay Key ID and Secret**, required to make secure API calls to Razorpay (for order creation, capturing payments, refunds, etc.).

📌 Why we used it?

- To safely read Razorpay credentials from the properties file.
- To avoid hardcoding sensitive information inside the code.
- To allow Spring services to inject and use Razorpay config easily.

📌 Benefits

- Better security
- Easy rotation of keys
- Cleaner code (single responsibility)

☑ 3. WebConfig.java

(CORS Configuration)

```
@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**")
                .allowedOrigins("http://localhost:5173")
                .allowedMethods("GET", "POST", "PUT", "DELETE")
                .allowedHeaders("*");
    }
}
```

📌 Purpose

This class configures **CORS (Cross-Origin Resource Sharing)** for your backend.

📌 Why we used it?

Your backend (Spring Boot) runs at:

```
http://localhost:8080
```

Your frontend (React Vite) runs at:

```
http://localhost:5173
```

These are **two different origins**, and browsers normally block such cross-origin requests.

So we enabled CORS to allow React to call Spring Boot APIs without getting blocked.

📌 What it allows?

- Allows requests from the React frontend
- Enables communication via GET, POST, PUT, DELETE
- Allows all headers

📌 Benefits

- Required for frontend-backend communication
- Fixes CORS errors
- Makes API usage safe and controlled

☑ 4. WebSocketConfig.java

(Real-time communication with STOMP WebSocket)

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void configureMessageBroker(MessageBrokerRegistry config) {
        config.enableSimpleBroker("/topic");
        config.setApplicationDestinationPrefixes("/app");
    }

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/ws")
                .setAllowedOriginPatterns("*")
                .withSockJS();
    }
}
```

📌 Purpose

This config enables **real-time communication** using WebSockets with STOMP protocol.

📌 Why we used it?

You are building a system (likely chat, order updates, restaurant status updates etc.) that requires **instant communication**, not normal HTTP requests.

Examples:

- Sending real-time chat messages
- Broadcasting order status updates
- Instant notifications
- Live updates for user actions

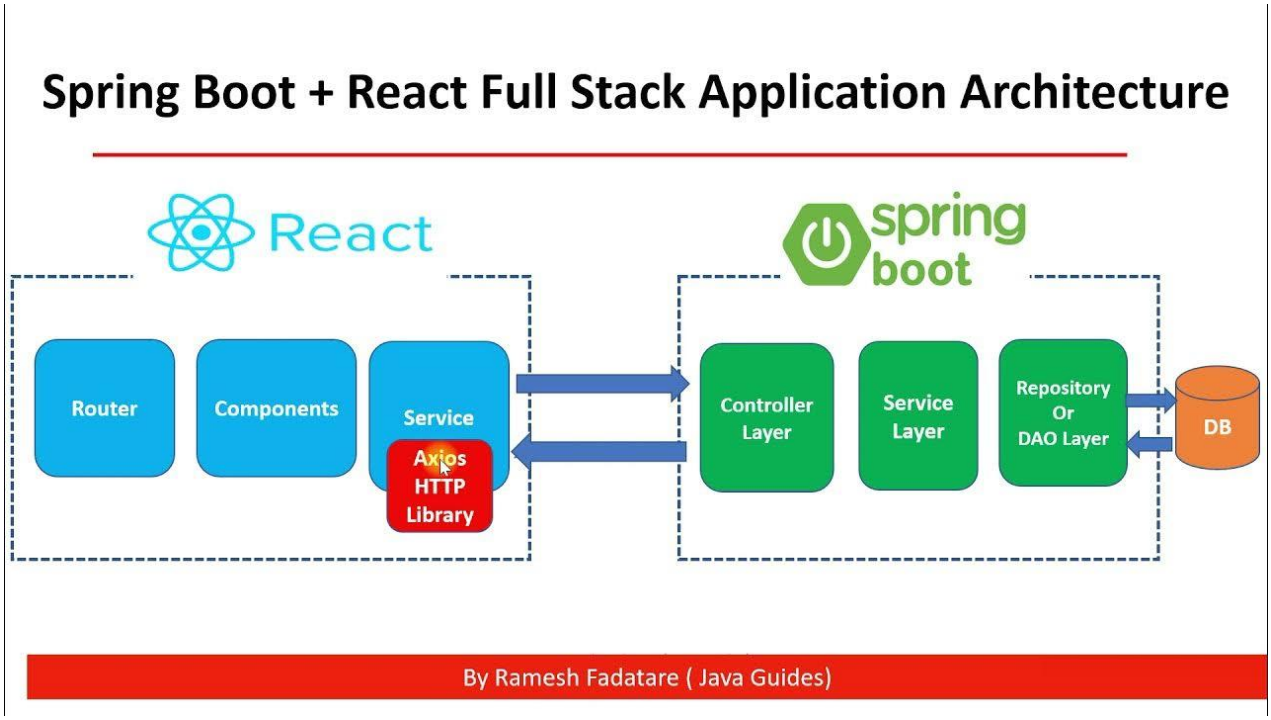
Key Features of WebSocketConfig

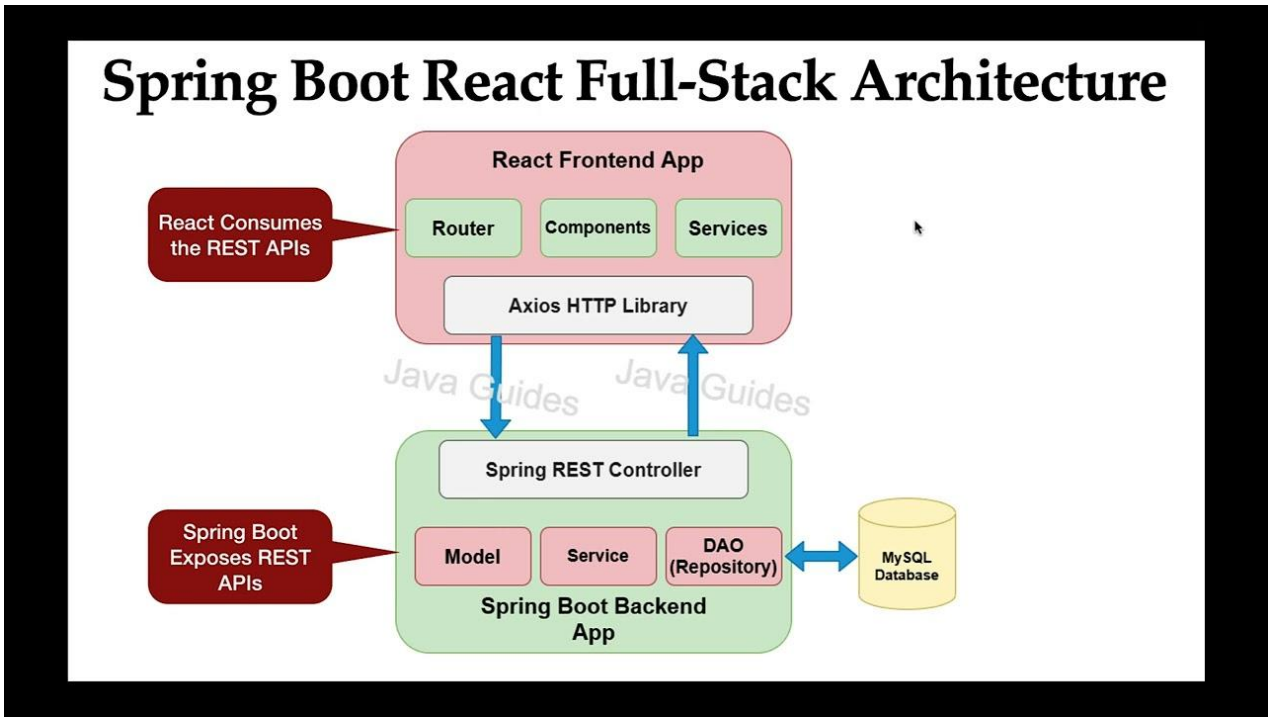
✓ `/ws` → Endpoint for clients to connect
✓ `/topic` → Broadcast messages to multiple users
✓ `/app` → Messages sent from client to server
✓ `.withSockJS()` → Fallback support for older browsers

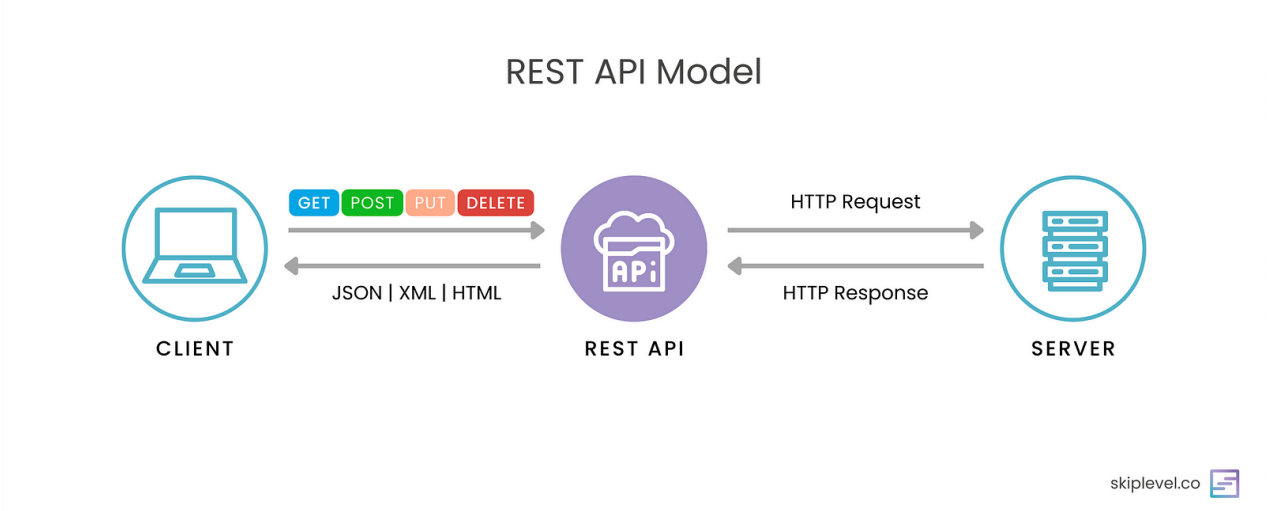
📌 Benefits

- Instant, bi-directional communication
- Better than polling or repeated API calls
- Reduced network load
- Modern architecture for chat and live systems

How Frontend Calls Backend Endpoints







system is built using:

- **Frontend:** React (Vite)
- **Backend:** Spring Boot REST API
- **Communication:** HTTP/HTTPS (REST API) + WebSocket for real-time

---

☑ 1. Frontend triggers an API call

The frontend uses **fetch()** or **axios** to call backend endpoints.

Example (React):

```
axios.get("http://localhost:8080/api/users");
```

This sends an HTTP request to your Spring Boot application.

☑ 2. CORS Allows the Request (Cross Origin Resource Sharing)

`WebConfig` allows the frontend domain:

```
.allowedOrigins("http://localhost:5173")
```

So browser says:

✓ Allowed → Send request
❌ If CORS was missing → Browser blocks the call

This step is **very important** when frontend & backend run on different ports.

☑ 3. Request Reaches Backend Controller

Spring Boot exposes REST endpoints using `@RestController` and `@RequestMapping`.

Example:

```
@RestController
@RequestMapping("/api/users")
public class UserController {

    @GetMapping
    public List<User> getAllUsers() {
        return userService.getUsers();
    }
}
```

When the frontend calls:
`GET /api/users`

Spring Boot maps it to this method.

☑ 4. Backend Processes the Request

Inside the controller:

1. Request is received
2. Spring validates it
3. Spring calls the service layer
4. Service interacts with database (via JPA/Hibernate)
5. Response is prepared

☑ 5. Backend Responds With JSON

Spring Boot automatically converts data into **JSON** using Jackson.

Example output:

```
[
  {
    "id": 1,
    "name": "Bharat",
    "email": "bharat@mail.com"
  }
]
```

☑ 6. Frontend Receives Response

React receives this JSON and uses it in UI.

Example:

```
const [users, setUsers] = useState([]);

useEffect(() => {
  axios.get("http://localhost:8080/api/users")
       .then(res => setUsers(res.data));
}, []);
```

Now the data is shown on UI using HTML/JSX.

★ Complete Request–Response Flow

React Component → Axios/Fetch → HTTP Request → Spring Boot Controller → Service → Database → JSON Response → React UI

Where Does WebSocket Fit? (For real-time messages)

WebSocketConfig enables:

- `/app` → client → server messages
- `/topic` → server → client broadcasts

Frontend example:

```
stompClient.subscribe("/topic/messages", handleMessage);
```

Backend example:

```
simpMessagingTemplate.convertAndSend("/topic/messages", message);
```

This is used for:

- Real-time chat
- Order status updates
- Notifications

IMP

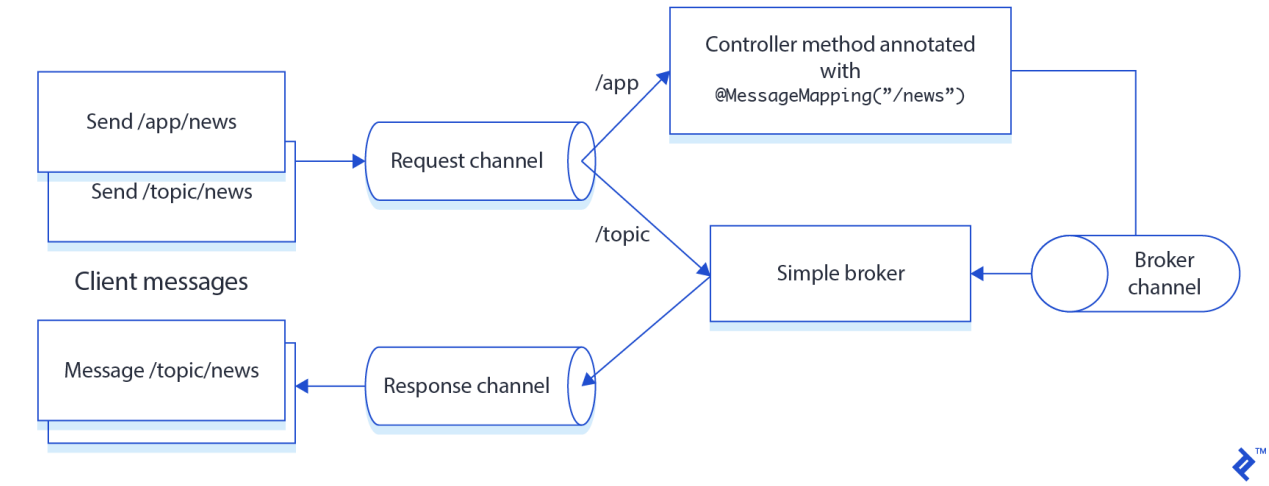
In this system, the React frontend communicates with the backend using REST APIs. The frontend calls backend endpoints using Axios or Fetch, which send HTTP requests to Spring Boot controllers. Spring handles the request, processes logic via service and repository layers, interacts with the database, and returns a JSON response.

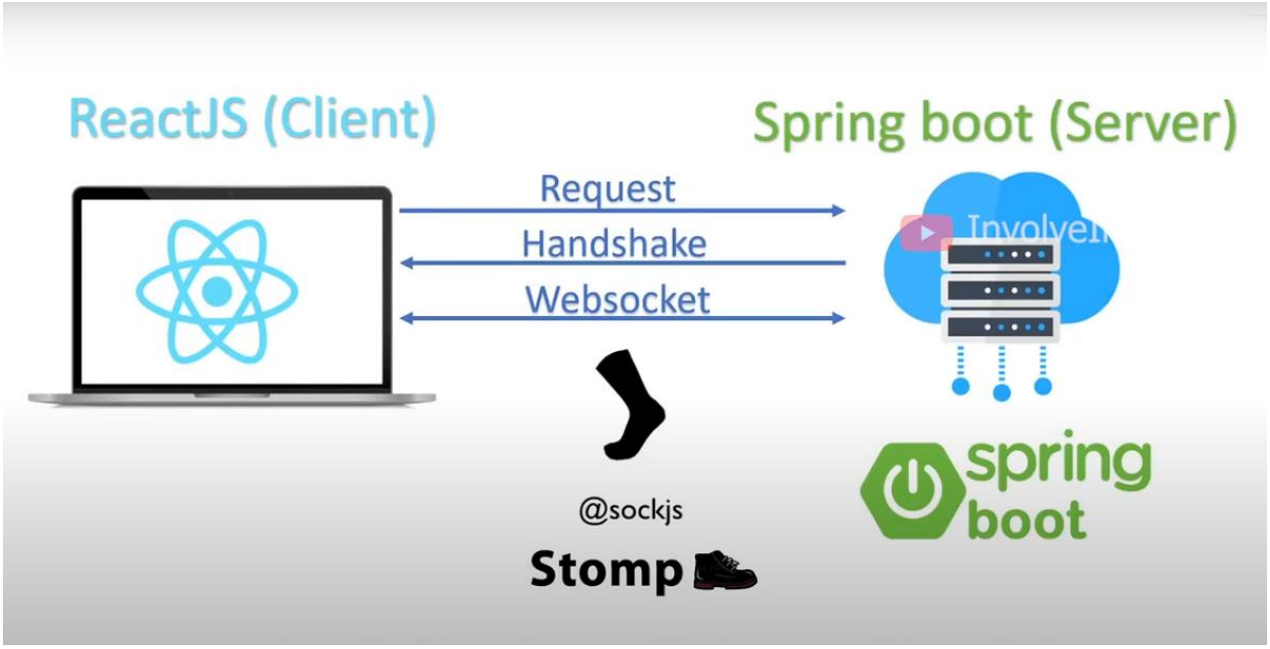
CORS configuration allows cross-origin access between React (5173) and Spring Boot (8080). For real-time features, we use WebSockets with STOMP, where the frontend subscribes to topics and receives instant updates from the backend.

The frontend interacts with the backend using REST APIs over HTTP/HTTPS. The React application triggers API calls using Axios or Fetch. These requests reach the Spring Boot application, where they are handled by REST controllers annotated with `@RestController` and specific endpoint mappings.

Spring Boot processes the request, calls the service layer, interacts with the database, and sends back the response in JSON format. The frontend receives this JSON and updates the UI accordingly. CORS configuration ensures that the React application running on a different port can communicate with the backend without browser restrictions. For real-time events, the system uses WebSockets with STOMP, enabling live communication between client and server.

🚀 How WebSockets Work for Real-Time Updates (Without Reload / Re-rendering)





WebSockets allow the frontend and backend to stay **connected continuously**, unlike REST APIs which work on request–response.

☑ 1. WebSocket Creates a Constant OPEN Connection

When your React app runs:

```
stompClient = Stomp.over(socket);
stompClient.connect({}, onConnected);
```

A **persistent connection** is opened between:

What happens internally:

- Browser opens a WebSocket connection to:
- `ws://localhost:8080/ws`
- Spring Boot WebSocketEndpoint (`/ws`) accepts connection
- A **persistent TCP pipeline** is created

No HTTP requests are needed afterwards.

- 🔹 Connection is now always ON
- 🔹 Backend can push data to frontend anytime

Frontend ↔ Backend

This connection stays alive until:

- Browser closes
- Component unmounts
- Network disconnects

There is **no need to send repeated requests**.

☑ 2. Frontend Subscribes to a Topic (Channel)

Example:

```
stompClient.subscribe("/topic/orderStatus", (msg) => {
    updateStatus(JSON.parse(msg.body));
});
```

This means:

✓ "Whenever backend posts anything on `/topic/orderStatus`, send it to me immediately."

📌 Multiple clients can subscribe to the same topic (broadcast).
📌 Each client maintains its own subscription callback.

Backend performs some business logic

Example: Order status changed, admin updated something, restaurant accepted order, etc.

A Spring service updates data:

```
order.setStatus("Accepted");
```

```
orderRepository.save(order);
```

☑ 3. Backend Pushes Updates Automatically

When backend wants to notify clients (React frontend), it uses:

Example:

```
simpMessagingTemplate.convertAndSend("/topic/orderStatus", update);
```

What happens internally:

- Spring Boot serializes `orderUpdateDto` into JSON
- Message is forwarded to **Message Broker** configured in:
- `config.enableSimpleBroker("/topic")`
- Broker forwards message to **all clients subscribed** to `/topic/order`

Whenever backend calls this line:

→ All connected clients instantly receive the message
→ No API call required
→ No page reload
→ No polling

This is called **Server Push**.

☑ 4. React Updates Only the Relevant Component (Not Whole Page)

In React, state is used to update UI:

```
const [status, setStatus] = useState("");

stompClient.subscribe("/topic/orderStatus", (msg) => {
    setStatus(JSON.parse(msg.body));
});
```

What happens internally?

- React sees that `setStatus()` changed only **one state variable**
- React triggers **a small re-render only for that component**
- Entire page is **not** re-rendered
- Only the element using `status` is updated

React updates state

```
setOrderStatus(update.status); or setStatus(JSON.parse(msg.body));
```

This is the key step.

React sees that the state variable changed.

✓ Only components using `orderStatus` will re-render
✓ The entire page DOES NOT reload
✓ Only a small part of the Virtual DOM updates

Example:

```
<p>Order Status: {status}</p>
```

Only this `<p>` tag updates — the rest of the UI does nothing.

This is why WebSockets feel "magical."

⭐ Why It Doesn't Re-render the Entire UI?

Because React uses:

✓ Virtual DOM

✓ Diffing Algorithm

React checks:

"What changed?"

If only one small value changed, only that part of the DOM is updated.

No:

- Reload
- Refresh
- Full re-render
- Polling
- API interval calls

🎯 Full Real-Time Flow

1. **WebSocket connection opens**
2. React subscribes to a topic
3. Backend publishes updates to that topic
4. WebSocket instantly delivers data to frontend
5. React updates state

6. React re-renders ONLY the affected component

This gives seamless real-time UI.

IMP

This application uses WebSockets to maintain a continuous, full-duplex connection between the frontend and backend. The React client establishes a persistent WebSocket connection and subscribes to specific STOMP topics. Whenever the Spring Boot backend performs an operation—such as updating an order status—it immediately publishes the update using `simpMessagingTemplate.convertAndSend()`. The Spring message broker then pushes this update to all subscribed clients in real time.

On the frontend, React receives the message instantly and updates only the specific component state instead of reloading the entire page. Thanks to React's Virtual DOM diffing mechanism, only the affected part of the UI re-renders, providing fast, smooth, and highly efficient real-time updates without repeated API calls or page refreshes. This end-to-end architecture ensures true real-time communication with minimal overhead and maximum performance.