📍 Research and Development of WhatsApp APIs: From Basics to Advanced.

During this phase, we focused on integrating WhatsApp APIs provided by **Meta** and setting up the infrastructure for seamless communication via WhatsApp for business purposes. Below are the key tasks and achievements:

- **Meta Developer Account Setup**: Created and configured the Meta Developer account for accessing WhatsApp Business APIs.
- **Access Token Generation**: Successfully generated a **permanent access token** for WhatsApp Business API integration.
- **WhatsApp Templates Creation**: Designed and customized WhatsApp templates tailored to client requirements for initiating business conversations. This included:
  - ✓ Creating templates for sending invoices in document format, text messages, images, videos, and audio messages.
  - ✓ Adding **dynamic variables** to templates for personalized content.
- **Testing with Test Numbers**: Before sending messages through the business account, we conducted extensive testing using a test number to ensure proper functionality.
- **Understanding Meta Policies**: Studied and implemented **Meta's privacy policies** to ensure compliant message sending, especially regarding:
  - ✓ **24-hour service window** for customer interactions.
  - ✓ **Business initiation messages** (messages sent without prior user consent).
- **Comprehensive Documentation**: Created detailed documentation for the various types of messages supported by the WhatsApp API, including text, images, videos, and invoices.
- **Advanced Work on WhatsApp API Integration and Chatbot Development**

In this phase, we took a deeper dive into the advanced features of **WhatsApp API** integration, focusing on creating a **dynamic and interactive chatbot** and implementing **webhooks** for real-time updates. Key tasks include:

- **Webhook Setup**:
  - ✓ Configured **webhooks** using **Node.js** and **Express.js** to track and monitor the status of WhatsApp messages, such as **sent**, **delivered**, and **read**.

- ✓ Enabled real-time updates for message status, ensuring better transparency and control over sent communications.

➢ **Automated Replies & Interactive Chatbot**:

- ✓ Built a **WhatsApp chatbot** that automates responses based on user interactions. The chatbot includes:
  - **Menu Options** – For users to select from predefined choices.
  - **Templates** – Used to send structured, reusable messages.
  - **Greeting Messages** – To initiate user interactions.
  - **Team Interaction** – Users can reach out to the team directly through the chatbot.
- ✓ Developed **interactive messages** that allow users to engage in dynamic conversations, improving customer experience.

➢ **Media Handling**:

- ✓ Integrated the ability to send and receive media via WhatsApp by uploading files to **Meta's server** and retrieving the **media ID** as a response.
- ✓ Ensured seamless integration of media content, such as images and videos, in user interactions with the chatbot.

➢ **WhatsApp API and Chatbot Integration**:

Implemented advanced functionality by integrating **WhatsApp Business API** with **chatbot logic**, ensuring smooth conversation flows and timely automated responses based on user inputs.

🤝 Team Collaboration — WhatsApp Receipt Integration with APIs in Mallikarjuna's project! (Document API)

As part of supporting our teammate's project (Mallikarjun's Event Booking), we contributed by **integrating the WhatsApp Cloud API** to automatically send **payment receipts** to customers once a transaction was successful.

🔧 **What We Did:**

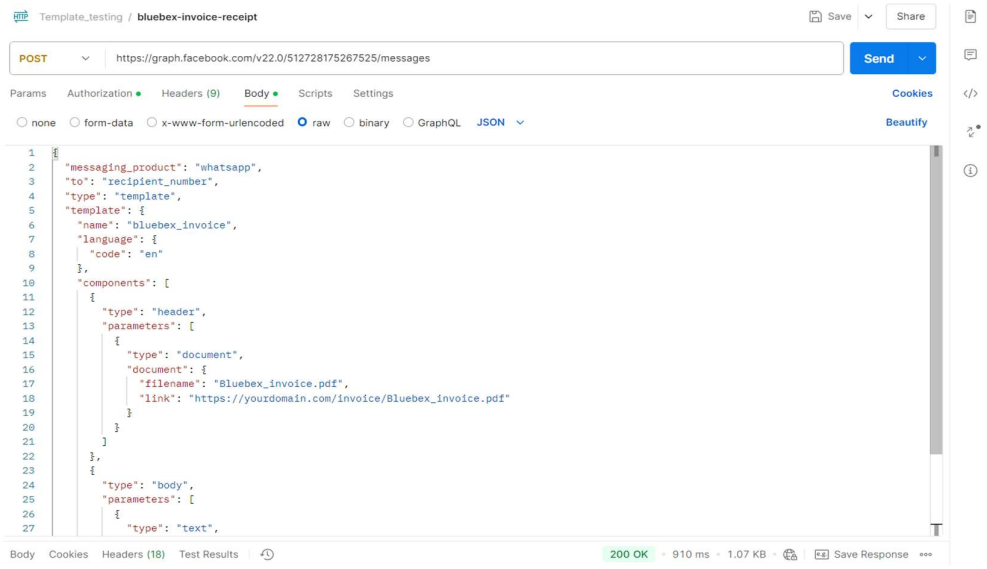
- **Created and configured** a WhatsApp Business account through **Meta Developer Portal**.
- **Generated a permanent access token** and obtained the **phone number ID**.
- Designed and submitted a **custom message template** (e.g., payment_receipt) to Meta for approval.
- Included **dynamic placeholders** (like customer name) inside the template.

APIS Integration:

- Used the **/messages** endpoint from Meta's Graph API to send template messages.
- Created the message payload including:
    - Customer's **WhatsApp number**
    - Approved **template name**
    - A **PDF attachment** of the invoice:
        - Sent either via **public URL** or **uploaded file (media ID)**.
- Automated this via **Node.js backend**, using **Axios** to call the API upon payment success.

Media Upload (PDF Invoice):

- Uploaded invoice PDFs using the **/media** endpoint to get a media_id.
- This media_id was included in the template's header to send the document directly on WhatsApp.



Result:

The system now automatically sends a **personalized message** and **PDF receipt** to the customer's WhatsApp after each successful payment, improving both professionalism and user satisfaction.

🌐 WhatsApp Cloud API — Message Types We Implemented

As part of the development process, we've created several APIs to send different types of messages via the WhatsApp Cloud API. These APIs handle everything from simple text messages to more interactive messages with buttons and lists.

Base URL for WhatsApp Cloud API:

https://graph.facebook.com/v18.0/PHONE_ID/messages

Replace PHONE_ID with your actual phone number ID.

Access token is required in authentication section in order to work all the APIs mentioned.

1. Text Message API

We implemented an API that sends a simple text message, which could be anything from a basic greeting to transactional information.

Example Request Body:

{

"messaging_product": "whatsapp",

"to": "RECIPIENT_PHONE_NUMBER",

"type": "text",

"text": {

"body": "Hello, this is a test message!"

}

}

2. Template Message API

For sending approved templates, such as order confirmations or payment receipts, we used this API. We customized the message with dynamic placeholders like the customer name or invoice number.

Example Request Body:

{

"messaging_product": "whatsapp",

"to": "RECIPIENT_PHONE_NUMBER",

"type": "template",

"template": {

"name": "TEMPLATE_NAME",

"language": {

"code": "LANGUAGE_CODE"

},

"components": [

{

"type": "body",

"parameters": [

{ "type": "text", "text": "John" }

]

}

]

}

}

3. Image with Message API

This API allows us to send media such as images via WhatsApp.

Example (Image Message) Request Body:

{

"messaging_product": "WhatsApp",

"to": "RECIPIENT_PHONE_NUMBER",

"type": "image",

"image": {

"link": "IMAGE_URL"

}

}

4. Document Messages API

We developed an API to send documents like invoices or receipts, either by linking to a file hosted online or uploading directly via the Meta API.

Example Request Body:

```
{
"messaging_product": "whatsapp",
"to": "RECIPIENT_PHONE_NUMBER",
"type": "document",
"document": {
"link": "DOCUMENT_URL",
"filename": "invoice.pdf"
}
}
```

5. Audio– Messages API

For sending audio files, we created an API that attaches an audio file through a URL.

Example Request Body:

```
{
"messaging_product": "whatsapp",
"to": "RECIPIENT_PHONE_NUMBER",
"type": "audio",
"audio": {
"link": "AUDIO_URL"
}
}
```

6. Contact Messages API

This API allows us to send contacts directly to users on WhatsApp, making it easy to share business cards or other contact information.

Example Request Body:

```
{
"messaging_product": "whatsapp",
"to": "RECIPIENT_PHONE_NUMBER",
"type": "contacts",
```

"contacts": [

{

"name": {

"formatted_name": "John Doe",

"first_name": "John",

"last_name": "Doe"

},

"phones": [

{

"phone": "1234567890",

"type": "WORK"

}

]}

]

}

7. Interactive Messages API (Buttons & Lists)

We also created APIs for sending interactive messages, like reply buttons and list messages. These are useful for user engagement and interaction.

Example (Reply Button) Request Body:

{

"messaging_product": "whatsapp",

"to": "RECIPIENT_PHONE_NUMBER",

"type": "interactive",

"interactive": {

"type": "button",

"body": {

"text": "Do you want to continue?"

},

"action": {

"buttons": [

{

"type": "reply",

"reply": {

"id": "yes_button",

"title": "Yes"} }] } }

}

📄 To upload any file and get the media_id and use in the APIs, use the following:

POST https://graph.facebook.com/v22.0/{phone_number_id}/media

🗜️ Form Data Parameters

file = Invoice98765432108.pdf

messaging_product = whatsapp

💡 Notes

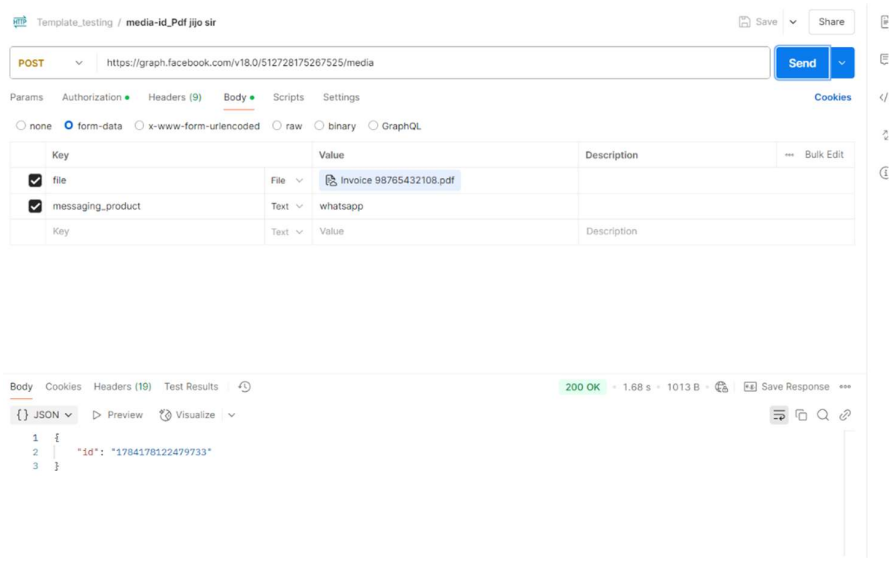
- Attach the file as form-data
- After uploading, you'll get a JSON response:

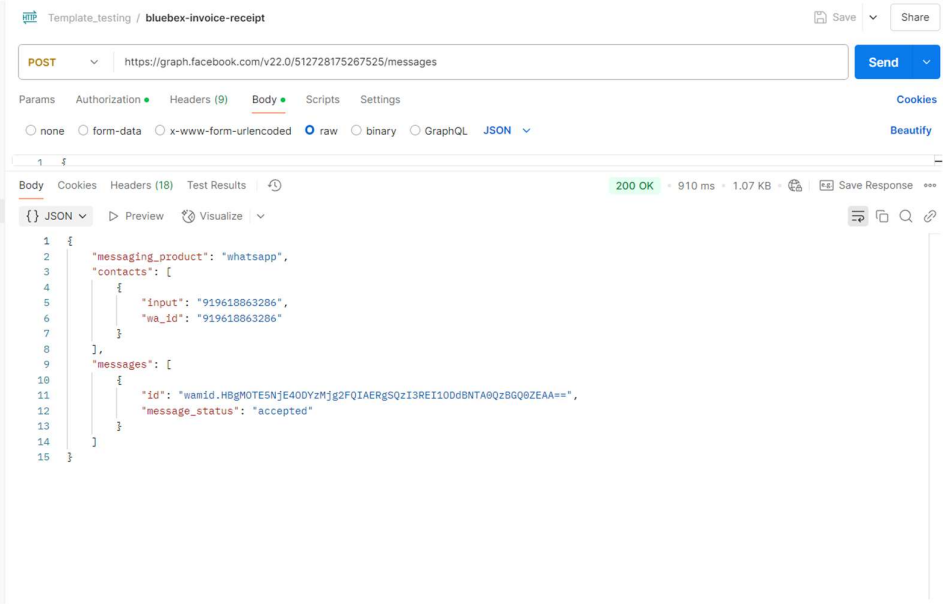
{

"id": "1393679178313783"

}

- Use this ID in your template message to send as a document





🚀 Milestones Achieved:

1. 📍 **Research and Development of WhatsApp APIs:** From account creation to webhook integration, we gained comprehensive knowledge of the WhatsApp Cloud API, enabling us to implement both basic and advanced messaging features.
2. 📝 **Creation of Message Templates:** Designed and submitted multiple approved message templates to initiate communication, allowing businesses to send messages without explicit user consent and automate confirmation processes.
3. 💬 **Implementation of Various Message Types:** Developed APIs for different message types, including text, template, interactive buttons, images, documents, and media attachments to enhance customer communication.
4. 📄 **Document API Integration:** Created solutions for sending and managing document attachments, helping our teammate ***mallikarjuna*** to use APIs to seamlessly integrate receipt and invoice functionalities in the project.
5. 🔄 **Webhook Integration:** Implemented webhooks for real-time updates, ensuring smooth and automated communication between the backend and WhatsApp Cloud API.

Project Bluebex Whatsapp Chatbot

🔹 Chatbot Foundation Setup

We commenced work on the **Bluebex WhatsApp Chatbot** project . The initial objective was to create a **simple automated chatbot** capable of responding to a user's greeting (Hi) with a professional welcome message.

As we have already worked on many APIs from WhatsApp to send different messages we wanted to implement a chatbot using the APIs by using webhooks.

🔹 Profile Name Extraction via Webhook

A key **milestone** was the **successful extraction of the WhatsApp user's profile name** using profile.name from the webhook payload. This allowed for **personalized messaging**, improving user engagement and setting the base for future CRM integrations.

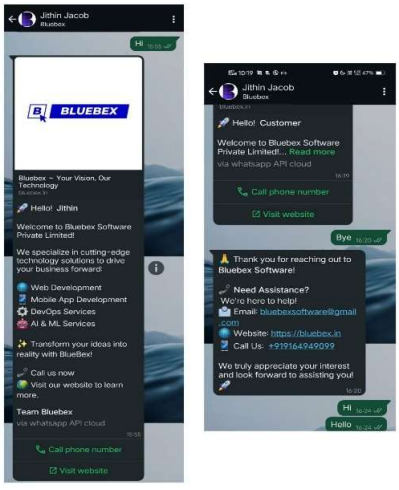
🔹 Webhook Integration

We successfully implemented **webhooks via WhatsApp Cloud API**, enabling real-time tracking of:

- Message delivery statuses (sent, delivered, read)
- Customer responses to our messages

This was critical in validating communication flow and planning for more dynamic bot logic.

Reference Image



Challenges Faced & Additional Work

✅ Deployment for Testing

Before facing the technical complexities, we **successfully deployed the chatbot to** Render.com for continuous testing and feedback collection. This allowed us to test API interactions in a live environment and monitor webhook responses efficiently.

📌 Understanding WhatsApp Messaging Rules

One of the most significant challenges in this phase was understanding the **WhatsApp Business Messaging Policies**, especially:

- **24-Hour Customer Service Window**: Initially, it was unclear why some messages—particularly interactive messages—were failing despite appearing correctly structured. I later understood that:
    - **Interactive messages** (buttons, list menus) are only allowed **after** a user sends a message, activating the 24-hour session.
    - **Before** this window, only **pre-approved message templates** can be used.

Project Bluebex Whatsapp Chatbot Enhancement

At the beginning of the second week of April, we **enhanced our WhatsApp Chatbot** built using the **WhatsApp Cloud API** to deliver a more interactive and engaging experience for users. Below are the key improvements and features:

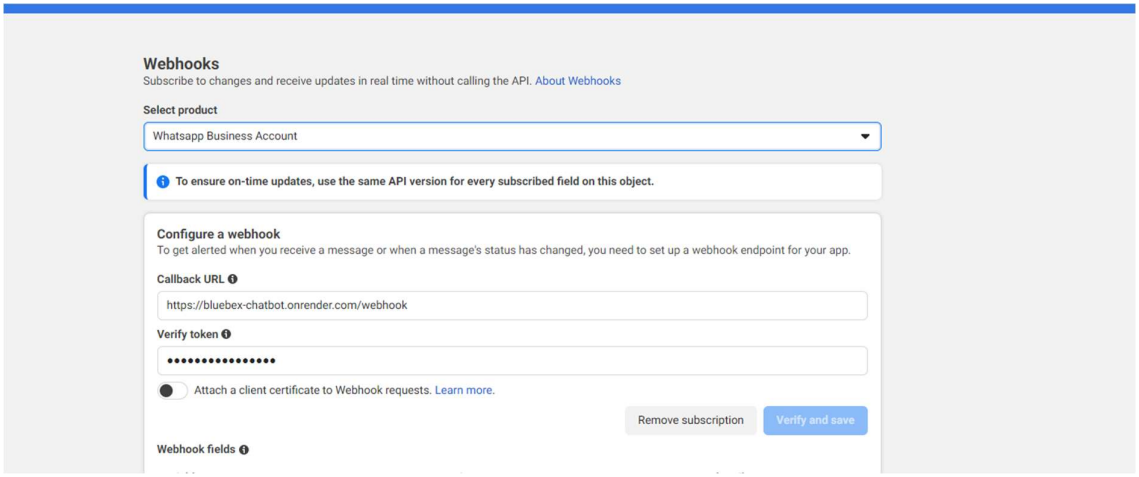
- The chatbot now **instantly responds** to greetings like "Hi" or "Hello" with a **personalized welcome template** and essential contact information.
- Introduced **service selection via interactive menus and buttons**, allowing users to explore:
    - Web Development
    - App Development
    - DevOps
    - AI & ML Services
- Based on user input, the bot now provides **detailed responses** and options such as **"Know More"** or **"Contact Us"**, enhancing user engagement.
- Developed using **Node.js and Express.js** with a clean, modular architecture and **detailed logging** for maintainability and debugging.
- **Chatbot deployed on Render.com** and available for testing via our business number **+91 74064 99990**—users can initiate the interaction simply by sending “Hi.”
- This upgrade marks a **significant transition** from a basic reply bot to an **intelligent, multi-step conversational assistant** ready for real-world use.

🔧 Backend Setup & WhatsApp Cloud API Integration

During the second week of April, we also focused on setting up the **backend infrastructure** required for our WhatsApp chatbot using **Node.js** and **Express.js**. The key objective was to integrate with the **WhatsApp Cloud API** and ensure smooth communication between our server and the WhatsApp platform.

Key Components Configured:

- **Meta App**: Created and configured on Meta for Developers.
- **Phone Number ID**: A unique identifier used to send and receive messages.
- **Permanent Access Token**: Used to authenticate API calls to WhatsApp.
- **Webhook URL**: An endpoint on our Express server to receive events from WhatsApp.
- **Verify Token**: A custom string set by us, used by Meta to validate the webhook during setup.



🔁 Webhook Verification (Simple Explanation)

When integrating with Meta's platform, Meta sends a special **challenge request** to our webhook URL during setup. Here's how it works:

1. Meta sends a **GET request** with parameters: hub.mode, hub.challenge, and hub.verify_token.
2. Our server checks if the verify_token matches the one we set in Meta Console.
3. If it matches, we send back the hub.challenge as a response — confirming the webhook is verified.

Once this verification is successful, Meta can start sending **message events** to our webhook.

Webhook Verification

```
app.get("/webhook", (req, res) => {
    const mode = req.query["hub.mode"];
    const challenge = req.query["hub.challenge"];
    const verifyToken = req.query["hub.verify_token"];

    if (mode === "subscribe" && verifyToken === VERIFY_TOKEN) {
        console.log("✅ Webhook Verified Successfully!");
        return res.status(200).send(challenge);
    }
    console.error("❌ Webhook Verification Failed!");
    res.sendStatus(403);
});
```

📩 Handling Incoming Messages

Once the webhook is active:

- We receive incoming messages in **JSON format** from WhatsApp users.
- Our server checks the message content and **extracts key details** like sender number and message body.
- Based on message type (e.g., "hi"), we respond appropriately—by sending a **list message** or a **custom reply**.
- 📋 Introduced **service selection via interactive menus and buttons**, allowing users to explore:
    - Web Development
    - App Development
    - DevOps
    - AI & ML Services
- 🧠 Based on user input, the bot now provides **detailed responses** and options such as **"Know More"** or **"Contact Us"**, enhancing user engagement.
- 🔄 This upgrade marks a **significant transition** from a basic reply bot to an **intelligent, multi-step conversational assistant** ready for real-world use.

Webhook Event Received Example:

Screenshots of the Bluebex WhatsApp chatbot

Project Name: WhatsApp Chatbot for BLUEBEX SOFTWARE PRIVATE LIMITED
Deployment Platform: Render.com (https://bluebex-chatbot-backend.onrender.com/)
Chatbot Access: Message **"hi"** to **+91 74064 99990**

---

1. Overview & Evolution of Progress

- The project began with a **fully functional chatbot** deployed earlier on Render, already tested and validated.
- Upon receiving a **completely new flow document from Jijo sir**, the team **swiftly transitioned** from maintenance to **active redevelopment mode**.
- Starting from the new flow document received, we:
    - **Reviewed the new document thoroughly**.
    - Understood the requirements.
    - Identified that a **significant code restructure** would be essential.
- The team responded with **quick adaptation and focused implementation**, ensuring consistency, stability, and cleaner architecture.

⚙️ 2. Development Workflow & Enhancements

- The new flow introduced **interactive WhatsApp messages**, replacing plain text prompts with:
    - **Button templates**
    - **List messages**
    - **Form-style lead collection**
    - **Templates**
- To achieve this:
    - Almost **every major section of the previous codebase was revisited** and optimized.
    - The **webhook logic was restructured** to handle dynamic flows, making it more modular and easier to maintain.
    - Extra attention was given to **clean coding practices**, **reusability**, and **performance consistency**.

---

3. Flow Updates – Key Enhancements

- **Software Development Services**:
    - Added interactive button-based selections for Android/Web/SaaS development.
    - Includes smart replies like “Talk to Team” and “Get a Quote”.
- **Internships & Courses**:
    - Created a new template for internships and courses once user wants to register interest, we send a form in the template collecting Full Name, Email, Educational Background, and Preferred Program.
- **Business Automation Tools**:
    - Integrated list message with sub-options (Invoicing App, WhatsApp Integration, Inventory Management, ERP, etc.)
    - Added follow-up CTAs: Demo, Pricing, Share Requirements.
- **Getting the client requirements**
    - Whenever user requests get quote regarding the pricing, we send a template attached with form to retrieve Full Name, Company Name, Project description, Timeline and Budget for project.
- **Talk to a Human**:
    - Provides direct interaction pathway with input capture and contact option.
- **Lead Capture Form**:
    - Created a unified form to gather relevant data when users request pricing, get a quote, register interest in internship.

---

4. Testing & Validation Phase

- **Extensive real-world testing** carried out via WhatsApp interface.
- **Webhook updates** are done manually in Meta Developer Console for every test iteration – this ensures live changes reflect instantly.
- **Observed behaviour**, adjusted text flows, and refined replies for better clarity and engagement.

- Minor content optimization is ongoing to further fine-tune responses.
- The chatbot is now in a **stable testing phase**, with deployment-ready structure.

---

5. Deployment & Accessibility

- Final code pushed to **Render.com** after testing:
  ✅ Deployment URL: https://bluebex-chabot-backend.onrender.com
  ✅ Test the bot by messaging: **+91 74064 99990**
- Live version reflects:
  - Full interactive message flow
  - Updated logic
  - Integrated templates and user inputs

---

6. Achievements & Next Steps

✅ **Achievements**

- Entire chatbot restructured within a **tight turnaround time**.
- Full integration of **new interactive flow** and **template-based interactions**.
- Clean, maintainable backend ready for scale and future additions.

📞 **User Experience Snapshot**

Greeting:

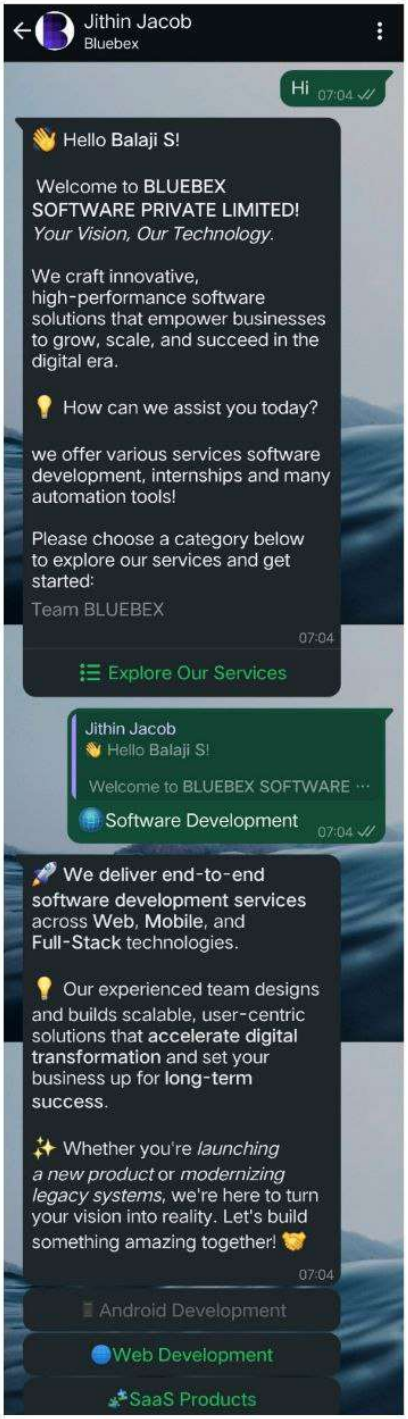
"Hi 👋, welcome to BLUEBEX SOFTWARE PRIVATE LIMITED – Your Vision, Our Technology..."

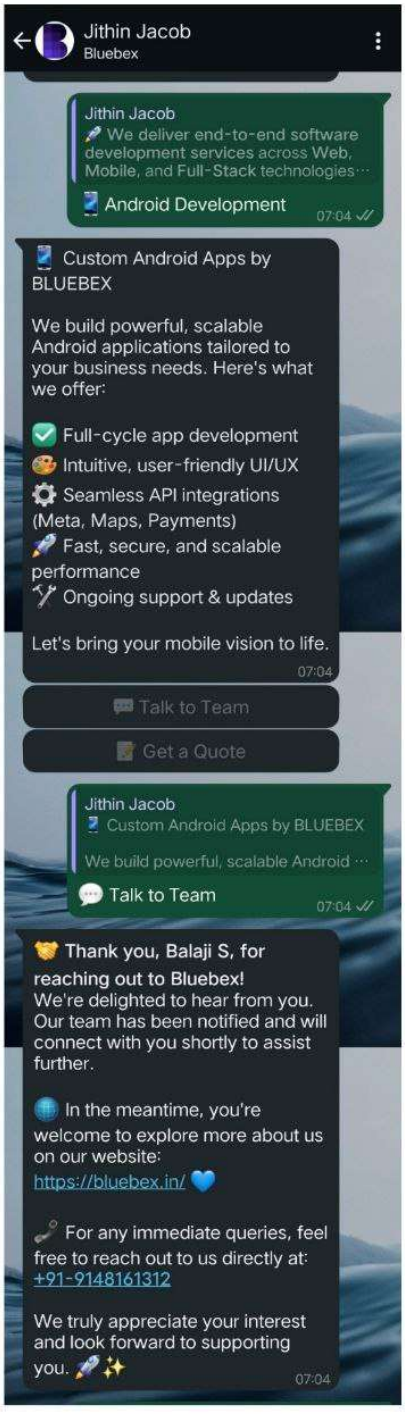
Final Closure Message:

"Thanks for reaching out to BLUEBEX! We'll get back to you within 24 hours. Meanwhile, check us out at https://bluebex.in"

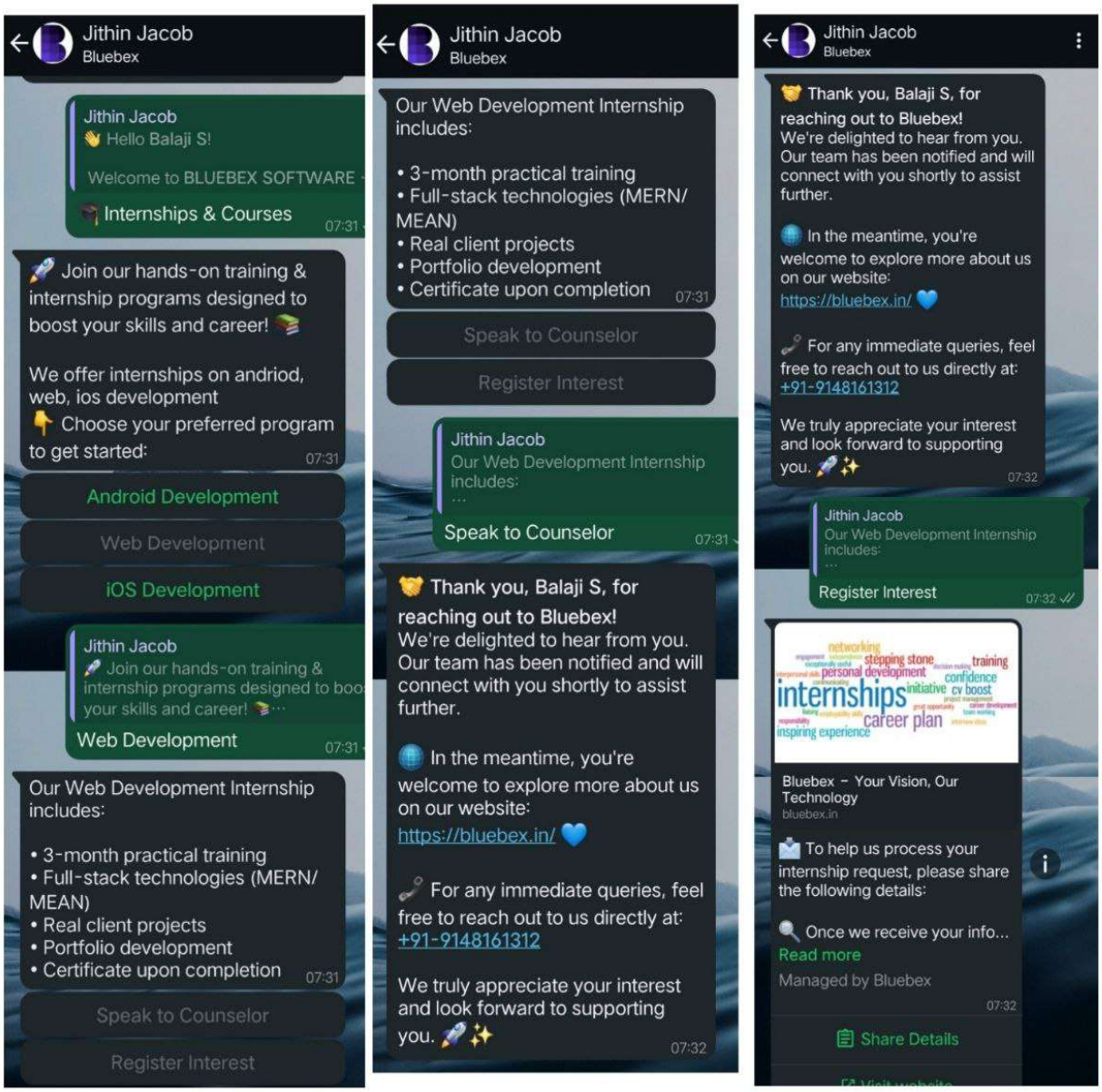
✓ **Some of the Screenshots**

Software development

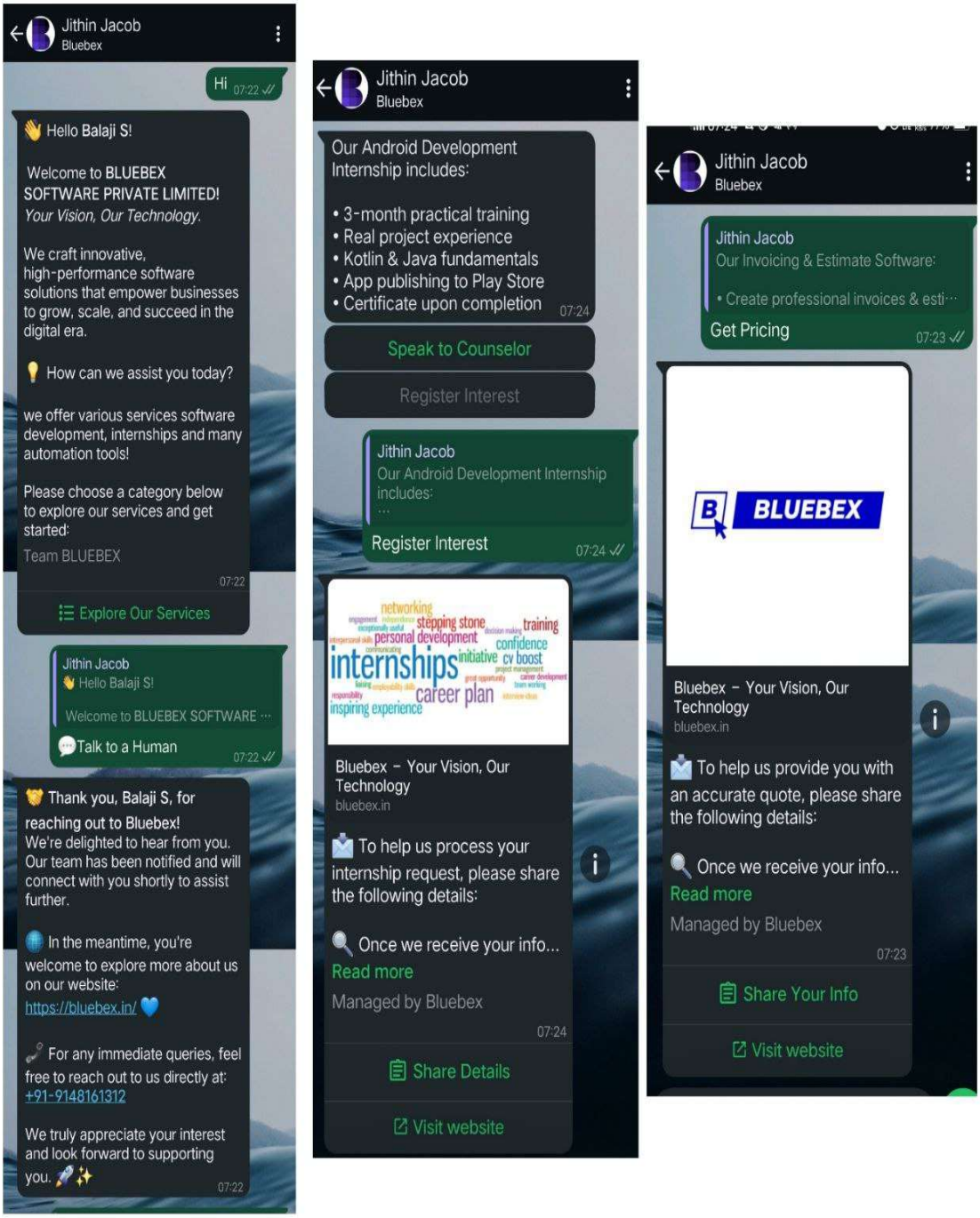




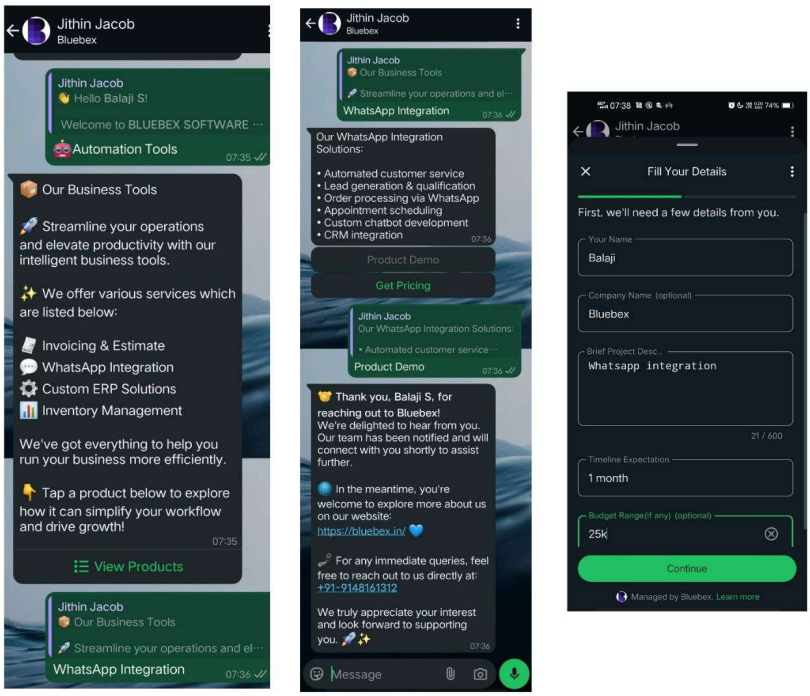
Internships and courses



Talk to a team and Get quote, Register Interest



Automation Tools



✅ **Key Milestones in Bluebex WhatsApp Chatbot.**

- 🚀 Successfully restructured the WhatsApp chatbot flow based on new requirements from Jijo Sir
- ⚙️ Implemented interactive message features: buttons, list messages, and lead collection forms
- 🔁 Integrated dynamic webhook logic to support modular and optimized flow management
- 🧪 Completed extensive real-world testing via WhatsApp interface for flow validation
- 🌐 Deployed the updated chatbot backend on Render: https://bluebex-chabot-backend.onrender.com
- 📞 Ensured live testing and debugging through WhatsApp: +91 74064 99990
- 💬 Enhanced user engagement with clear, concise, and interactive message flows
- 📄 Finalized lead capture forms for internship registration, quote requests, and product inquiries
- ✅ Achieved a stable, fully operational chatbot now ready for deployment

---

🔧 Step 1: Configure Firebase Admin SDK with Node.js

To enable communication between your WhatsApp chatbot and Firebase Realtime Database, you need to configure the Firebase Admin SDK in your Node.js application.

✅ Prerequisites

Before proceeding, ensure you've:

- Created a Firebase project in the Firebase Console
- Generated a Service Account key from Project Settings > Service accounts
- Installed the Firebase Admin SDK

📦 Install Firebase Admin SDK

Run the following command in your project directory:

```
npm install firebase-admin
```

📁 Setup Environment Variables

Create a .env file in the root of your project and define the following variables:

FIREBASE_PROJECT_ID=your-project-id

FIREBASE_CLIENT_EMAIL=your-service-account-email@your-project-id.iam.gserviceaccount.com

FIREBASE_PRIVATE_KEY="-----BEGIN PRIVATE KEY-----\nYOUR_PRIVATE_KEY\n-----END PRIVATE KEY-----\n"

FIREBASE_DATABASE_URL=https://your-project-id.firebaseio.com

```
require("dotenv").config();

const config = {
  port: process.env.PORT || 3000,
  whatsappToken: process.env.TOKEN,
  verifyToken: process.env.VERIFY_TOKEN,
  whatsappPhoneNumberId:
  process.env.WHATSAPP_NUMBER_ID || process.env.PHONE_NUMBER_ID,
};

module.exports = config;
```

📄 Firebase Admin Initialization Code (config/firebase.js)

```
// Initialize Firebase Admin
const initializeFirebase = () => {
  try {
    if (!admin.apps.length) {
      admin.initializeApp({
        credential: admin.credential.cert({
          projectId: process.env.FIREBASE_PROJECT_ID,
          clientEmail: process.env.FIREBASE_CLIENT_EMAIL,
          privateKey: process.env.FIREBASE_PRIVATE_KEY?.replace(/\\n/g, "\n"),
        }),
        databaseURL: process.env.FIREBASE_DATABASE_URL,
      });
      logger.info("Firebase Admin initialized successfully");
    }
    return admin.database();
  } catch (error) {
    logger.error("Error initializing Firebase Admin:", error);
    throw error;
  }
};

const db = initializeFirebase();
```

✓ Outcome

Once this is set up, your application will be connected to Firebase Realtime Database, and you can now read/write WhatsApp chatbot messages in real time.

✓ Summary – Firebase Admin SDK Setup for WhatsApp Chatbot

1. Installed firebase-admin package to connect Node.js with Firebase services.
2. Generated a Firebase Service Account key for secure authentication.
3. Stored sensitive credentials (project ID, client email, private key, database URL) in .env file.
4. Initialized Firebase Admin SDK using environment variables in a dedicated config file.
5. Connected to Firebase Realtime Database to store WhatsApp chatbot messages.
6. Enabled real-time read/write operations for incoming and outgoing messages.

✓ How Webhooks Are Verified

1. Webhook URL Setup
   Meta (or any platform) sends a verification request to your webhook URL when you set it up in the developer dashboard.
2. Incoming Request Parameters
   The verification request contains three query parameters:
   - hub.mode (should be 'subscribe')
   - hub.challenge (random string you must return)
   - hub.verify_token (must match your own secret token)
3. Token Matching
   Your server compares the received hub.verify_token with the one you defined (e.g., in config.verifyToken).
4. Successful Verification
   If hub.mode === "subscribe" and the verify token matches, the server:
   - Logs success
   - Responds with the hub.challenge (as required by Meta)
5. Failed Verification
   If the token doesn't match or the mode is incorrect:
   - Server logs an error
   - Responds with status code 403 (Forbidden)
6. Purpose
   This verification ensures that only authorized services (like WhatsApp Business API) can connect to your webhook.

```
//Verify webhook
const verifyWebhook = (req, res) => {
  const mode = req.query["hub.mode"];
  const challenge = req.query["hub.challenge"];
  const verifyToken = req.query["hub.verify_token"];

  if (mode === "subscribe" && verifyToken === config.verifyToken) {
    logger.success("Webhook Verified Successfully!");
    return res.status(200).send(challenge);
  }
  logger.error("Webhook Verification Failed!");
  return res.sendStatus(403);
};
```

📦 WhatsApp Webhook Processing – Documentation

This function handles incoming POST requests from WhatsApp's webhook, which may include message events or message status updates. Below is a breakdown of how the logic works.

✅ 1. Log and Validate Incoming Webhook Payload

```
const body = req.body;
logger.webhookEvent(body);

if (!body.object) return res.sendStatus(404);
```

- Logs the raw webhook body for debugging and auditing.
- Checks if the payload contains the expected structure (body.object). If missing, returns HTTP 404.

✅ 2. Extract Message Changes

```
const changes = body.entry?.[0]?.changes?.[0]?.value;
const phone_number_id = changes?.metadata?.phone_number_id;
```

- Extracts nested data from the webhook payload.
- Retrieves the phone_number_id which is needed to send replies through WhatsApp API.

✅ 3. Handle Message Status Updates

```
const statuses = changes?.statuses;
if (statuses && statuses.length > 0) {
  const statusInfo = statuses[0];
  const status = statusInfo.status;
  const recipientId = statusInfo.recipient_id;
  const messageId = statusInfo.id;

  logger.info(`📬 Status update received: ${status} for ${recipientId}`);

  return res.sendStatus(200); // Early exit
}
```

- Handles status updates like sent, delivered, or read.
- Extracts key details: status type, recipient ID, message ID.
- Logs the update and exits early after handling.

✅ 4. Extract Incoming Message

```
const messages = changes?.messages;

if (!messages || !phone_number_id) return res.sendStatus(200);

const message = messages[0];
const from = message.from;
const messageType = message.type;
const profileName = changes?.contacts?.[0]?.profile?.name || "User";
logger.info("Extracted profile Name", profileName);
```

- Checks if an actual message exists and phone_number_id is valid.
- Extracts:
    - Message sender's WhatsApp number (from)
    - Message type (text, interactive, etc.)
    - User profile name (for personalization)

✅ 5. Handle Interactive Messages

```
if (message.type === "interactive") {
  await handleInteractiveMessage(phone_number_id, from, message, profileName);
}
```

- If the user sends a button reply or list selection, it's considered interactive.
- Calls a separate function handleInteractiveMessage() to manage it.

✅ 6. Handle Text Messages

```
else if (message.text) {
  await handleTextMessage(phone_number_id, from, message.text.body, profileName);
}
```

- For simple text inputs, calls handleTextMessage() with message content.
- Ideal for handling greetings, queries, or default responses.

✅ 7. Error Handling

```
} catch (error) {
  logger.error("Error processing webhook", error);
  return res.sendStatus(500);
}
```

- Catches unexpected issues and logs them.
- Returns HTTP 500 to indicate a server-side error.

In Our Backend we use different types of message handlers to send various formats of messages, each suited for specific purposes and interactions:

1. **templateHandler**
    - Used exclusively for WhatsApp-approved message templates.
    - Ideal for automated outbound notifications like OTPs, appointment reminders, payment confirmations, or order updates.
    - These templates follow strict formatting rules approved by WhatsApp, ensuring consistent and trustworthy communication.
2. **messageHandler**
    - Handles basic plain text messages or messages combined with simple interactive elements like buttons.
    - Commonly used for sending informative messages with quick reply buttons such as “Talk to Counselor” or “Register Now.”
    - Useful for one-step user interactions and delivering straightforward content.
3. **interactiveHandler**
    - Supports rich, dynamic message formats designed to enhance user engagement and simplify navigation.

- Includes two primary interactive message types:
    - List Messages: Present users with multiple selectable options in a scrollable list. Great for menus or service selections.
    - Button Messages: Display up to three reply buttons for quick user responses, perfect for limited-choice questions or call-to-action prompts.

☑ What Each Function Does & Which Handler It Uses

Function Name	Handler Type Used	Description
sendListMessage	interactiveHandler	Sends a **welcome list message** presenting multiple service categories (like Software, Internship, Automation). Enables users to easily select what interests them.
sendSoftwareDevOptions	interactiveHandler	Sends a **button message** with options for different software development services (Android, Web, SaaS). Designed for quick, one-tap selections.
sendInternCoursesOptions	interactiveHandler	Sends a **button message** with internship course options (Android, Web, iOS). Facilitates fast, interactive course selection.
sendAutomationToolsOptions	interactiveHandler	Sends a **list message** that displays various business automation tools offered, allowing users to browse and pick according to their needs.
handleInternshipRegisterInterest	templateHandler	Sends a **pre-approved WhatsApp template message**, often used to confirm interest or send a quote/request form to the user. Ensures compliance with WhatsApp's messaging policies.
sendInternshipDetails	messageHandler	Sends detailed text about internships along with **interactive buttons** (e.g., "Talk to Counselor," "Register") to guide users toward further action.
sendProductDetails	messageHandler	Sends comprehensive product or service information, combined with **buttons** for actions like inquiry or purchase, making it easy for users to engage immediately.

Documentation: Handle Interactive Message

Purpose

The handle Interactive Message function processes incoming interactive messages from users, including:

- Form submissions (nfm_reply)
- Standard interactive messages (button clicks, list selections)
- Routes each interaction to appropriate handlers or sends predefined responses

It also logs actions and stores messages for auditing and further processing.

Function Signature

async function handleInteractiveMessage(phoneNumberId, from, message, profileName)

Parameters:

- phoneNumberId (string): The WhatsApp Business API phone number ID used for sending messages.
- from (string): The sender's phone number (user).
- message (object): The interactive message payload received.
- profileName (string): The profile name of the user sending the message.

Overview of Handling Types

1. Handling nfm_reply (Form Submissions)

- Detects if the interactive message is a form submission (message.interactive.type === "nfm_reply").
- Parses the JSON form response inside message.interactive.nfm_reply.response_json.
- Stores the form data in the chat service (ChatService.storeChatMessage).
- Checks for specific flow tokens to differentiate form types and handle accordingly:
    - Example: If flow_token is "unused", treat it as an internship registration form:

        - Sends a thank-you confirmation message.
        - Forwards the conversation to a human agent.
- On JSON parsing errors, sends an error message to the user.

2. Handling Standard Interactive Messages (Buttons, Lists, etc.)

- Extracts the selected option ID from the interactive message.
- If no selected ID is found, sends an error message and stops.
- Stores the selected interactive response in chat service.
- Uses a switch statement on selectedId to route handling:

Main Menu Options

- "software_dev": Sends software development sub-options.
- "intern_courses": Sends internship courses options.
- "automation_tools": Sends automation tools options.
- "talk_human": Transfers the user to a human agent.

Software Development Sub-options

- "android_dev": Sends details about Android app development plus buttons to talk to the team or get a quote.
- "web_dev": Sends details about web development with similar call-to-action buttons.
- "saas_products": Sends SaaS products info with call-to-action buttons.

Internship Options

- "intern_register": Handles internship registration interest.
- "intern_android", "intern_web", "intern_ios": Sends detailed internship info for selected domains.

Automation Tools Options

- "invoicing_app", "whatsapp_integration", "custom_erp_solutions", "inventory_management": Sends product details for selected automation tools.

Call-to-action Buttons within Sub-menus

- "talk_team": Sends a message to connect user to a human agent.
- "get_quote": Sends a quote request form.
- "internship_quote": Sends a quote form specific for internships.

Default Handling

- For unrecognized IDs, logs the event.
- Sends a generic thank-you message.
- Transfers to human agent for further assistance.

3. Error Handling

- Catches any errors during processing.
- Logs the error.
- Sends an error message to the user indicating a processing problem.
- Transfers the user to a human agent for help.

Summary Table of selectedId Handling

selectedId	Handling Action
software_dev	Send software dev options
intern_courses	Send internship course options
automation_tools	Send automation tools options
talk_human	Connect to human agent
android_dev	Send Android dev info + buttons
web_dev	Send Web dev info + buttons
saas_products	Send SaaS product info + buttons
intern_register	Handle internship registration interest
intern_android	Send internship details for Android
intern_web	Send internship details for Web

selectedId	Handling Action
intern_ios	Send internship details for iOS
invoicing_app	Send invoicing app product details
whatsapp_integration	Send WhatsApp integration product details
custom_erp_solutions	Send custom ERP solutions product details
inventory_management	Send inventory management product details
talk_team	Connect to human agent
get_quote	Send quote request form
internship_quote	Send internship quote form
Default / Unknown ID	Send generic thanks + transfer to human

Logging

- Logs interactive message type.
- Logs full interactive message JSON for debugging.
- Logs parsed form data.
- Logs unhandled interactive IDs.
- Logs errors with descriptive messages.

Example Flow for Form Submission (nfm_reply)

1. User submits form → message.interactive.type is "nfm_reply".
2. Form JSON is parsed.
3. Form data stored.
4. If flow_token matches "unused":
    - Sends confirmation message.
    - Forwards chat to human.
5. If parsing fails → sends error message.

---

1. handleInteractiveMessage()

Purpose:
Handles interactive messages received via WhatsApp Business API, such as buttons, list replies, and native form (nfm) submissions.

Function Signature

const handleInteractiveMessage = async (phoneNumberId, from, message, profileName)

Parameters

Parameter	Type	Description
phoneNumberId	String	WhatsApp Business phone number ID
from	String	WhatsApp user's phone number (sender)
message	Object	Full message object from WhatsApp
profileName	String	User’s profile name from WhatsApp

Main Logic & Handling Types

1. 🔁 nfm_reply (Form Submission)

- Parses response_json data from the form submission.
- Saves form submission to database.
- Sends confirmation message to user.
- Forwards to human agent (if needed based on flow_token).

Example Flow:

```
{
 "type": "nfm_reply",
 "nfm_reply": {
  "response_json": "{\"flow_token\": \"unused\", \"name\": \"John\"}"
 }
}
```

2. 📋 Standard Interactive Types

Handles interactive messages of types: button_reply, list_reply, etc.

Logic:

- Extracts selectedId from user interaction.
- Stores user selection in the database.
- Responds to the selected ID accordingly.

Supported IDs:

ID	Description
software_dev	Sends software development options
intern_courses	Sends available internship courses
automation_tools	Sends automation tools options
talk_human	Connects to human agent
android_dev	Sends Android dev service details
web_dev	Sends web dev service details
saas_products	Sends SaaS products list
intern_register	Initiates internship registration
intern_android	Sends Android internship details
intern_web	Sends Web internship details
intern_ios	Sends iOS internship details
invoicing_app	Product info: Invoicing App
whatsapp_integration	Product info: WhatsApp API Integration
custom_erp_solutions	Product info: Custom ERP
inventory_management	Product info: Inventory System
talk_team	Talk to BLUEBEX team
get_quote	Opens Quote Form

ID	Description
internship_quote	Opens Internship Quote Form

Error Handling

- ❌ If parsing fails or selectedId is missing, fallback message is sent.
- General try/catch block logs and forwards user to human agent in case of any exceptions.

✅ **2. handleTextMessage()**

Purpose:
Handles incoming plain text messages from WhatsApp users and responds based on keyword detection.

Function Signature

const handleTextMessage = async (phoneNumberId, from, messageText, profileName)

Parameters

Parameter	Type	Description
phoneNumberId	String	WhatsApp Business phone number ID
from	String	WhatsApp user's phone number (sender)
messageText	String	Plain text message sent by the user
profileName	String	User’s profile name from WhatsApp

---

1. 💬 Text Normalization

- Trims and lowercases the message for easier matching.

2. 🔎 Keyword/Greeting Detection

- Compares user input against a list of predefined greeting/trigger phrases.
- If a match is found, it responds with a list menu of options.

Sample trigger keywords:

hi, hello, hey, good morning, services, bluebex, bot, share your services

3. 🔁 Default Fallback

- If message doesn't match any known keywords:
    - Forwards the user to a human agent with a polite message.

✅ Summary Table of Handlers

Message Type	Handler Function	Trigger	Output
Text Message	handleTextMessage()	Greeting or service-related text	Sends list or connects to human agent
Interactive Message	handleInteractiveMessage()	Button/List/Form interaction	Sends options, product info, or opens forms
Form Submission	(inside handleInteractiveMessage)	nfm_reply	Parses form and sends confirmation + forwards if needed

---

📘 ChatService.storeChatMessage Documentation

Overview

The ChatService.storeChatMessage() method is responsible for storing incoming or outgoing chat messages (including form submissions) into **Firebase Realtime Database** under a structured path based on the sender's phone number. It also records user-level metadata such as profileName and phoneNumber.

This is typically used in systems that interact with chat platforms like WhatsApp, Messenger, or a custom frontend to persist user interactions.

🔧 Method Signature

```js
static async storeChatMessage(message)
```

📥 Parameters

Name	Type	Required	Description
`message`	Object	✅ Yes	The message object containing chat details to store.

📄 Expected `message` Object Format

```js
{
  content: "Hello there!",
  profileName: "John Doe",
  phoneNumber: "1234567890",
  messageType: "text" | "image" | "form_submission" | ...,
  direction: "incoming" | "outgoing", // optional
  timestamp: "2025-05-23T10:20:30Z" // optional
}
```

- `direction` defaults to `"incoming"` if not provided.
- `timestamp` defaults to the current time in ISO format if not provided.

📂 Firebase Database Structure

Each user's messages are stored under their phone number in the following structure:

```css
chats/
  └── 1234567890/
      ├── phoneNumber: "1234567890"
      ├── profileName: "John Doe"
      └── messages/
          ├── -N123abc...: { content, messageType, direction, ... }
          ├── -N124def...: { ... }
```

🧠 Behavior Logic

1. **Form Submissions:**
    - If messageType is "form_submission", the function bypasses typical formatting and directly pushes the message to:

chats/{phoneNumber}/messages/

This is because form submissions may contain complex or nested structures.

Standard Messages:

- Constructs a well-structured messageData object including:
    - content
    - profileName
    - phoneNumber
    - messageType
    - direction (defaults to "incoming")
    - timestamp (defaults to new Date().toISOString())
    - createdAt (added to track creation time in the backend)

User Metadata:

- Updates user-level data:
    - chats/{phoneNumber}/profileName
    - chats/{phoneNumber}/phoneNumber

Stores the Message:

- Pushes the messageData object to:

chats/{phoneNumber}/messages/

✅ **Success Logging**

- Logs message storing status using:

logger.info(`Stored message for ${message.phoneNumber} with all fields.`);

For form submissions:

logger.info(`Storing form submission for ${message.phoneNumber}`);

❌ **Error Handling**

- Wraps the logic in a try...catch block.
- Logs errors using:

logger.error("Error storing chat message:", error);

🔁 **Example Usage**

await ChatService.storeChatMessage({

content: "I'd like to know more about your services.",

profileName: "Jane Smith",

phoneNumber: "9876543210",

messageType: "text",

});

📄 **Outgoing Message Handling – Documentation (Text Messages)**

🎯 **Objective**

Enable sending of outgoing **text messages** via **WhatsApp Cloud API**, and **store them in Firebase Firestore** with metadata such as message type and direction.

✅ **Scope**

This document covers **only outgoing text messages**. Interactive (buttons/lists) and template messages will be handled next.

📁 File:Services /messageHandler.js

🔧 Function: sendTextMessage(phoneNumberId, recipient, message, profileName = "Bluebex")

🧩 Description

Sends a plain text message to a WhatsApp user using the WhatsApp Cloud API and logs the outgoing message into Firebase Firestore through ChatService.storeChatMessage().

🔄 Flow Breakdown

1. ✅ Send Message via WhatsApp API

```
js                                                    Copy    Edit

await axios.post(
  `https://graph.facebook.com/v22.0/${phoneNumberId}/messages`,
  {
    messaging_product: "whatsapp",
    to: recipient,
    type: "text",
    text: { body: message },
  },
  {
    headers: {
      "Content-Type": "application/json",
      Authorization: `Bearer ${config.whatsappToken}`,
    },
  }
);
```

2. 🪵 Log Success

logger.success(`Text message sent to ${recipient}`);

3. 💾 Store Message in Firestore

```
await ChatService.storeChatMessage({
  content: message,
  profileName,
  phoneNumber: recipient,
  messageType: "outgoing_text",
  direction: "outgoing",
});
```

🗂️ Firestore Data Format

Field	Description
`content`	Message text sent to the user
`profileName`	Name of sender (default: "Bluebex")
`phoneNumber`	WhatsApp number of the recipient
`messageType`	Always `"outgoing_text"` for text messages
`direction`	Always `"outgoing"`

✅ Key Milestones Achieved

✅ Task	Description
WhatsApp API Integration	Using `axios` with secure token authorization
Outgoing Text Message Logged	Stored in Firestore with metadata
Message Type Tracked (`outgoing_text`)	Helps differentiate between text, button, etc.
Direction Tracked (`outgoing`)	Helps separate sent messages from received ones
Modular & Reusable Logic	Easy to extend to other message types

Outgoing Interactive Message Handling Documentation

🎯 Objective

To send various **interactive WhatsApp messages** (list messages, button messages) via WhatsApp Cloud API and store them in Firebase Firestore with metadata that tracks the message type and direction.

📁 Overview of Files and Flow

- **services/interactiveHandler.js**
  Handles the storage of all outgoing interactive messages to Firebase Firestore with the proper message metadata.
- **services/chatService.js**
  Provides the reusable storeChatMessage() method used to save messages into Firestore.

📄 How Outgoing Interactive Messages Are Sent and Stored

1. Sending Interactive Messages

- Functions such as sendListMessage(), sendSoftwareDevOptions(), sendInternCoursesOptions(), and sendAutomationToolsOptions() are designed to:
    - Construct the appropriate WhatsApp interactive message payload according to the message type:
        - **List messages:** with sections, rows, button text, etc.
        - **Button messages:** with multiple reply buttons.
    - Send the message via axios.post() to the WhatsApp Cloud API endpoint:

    https://graph.facebook.com/v22.0/${phoneNumberId}/messages

    - Include authorization headers with the WhatsApp Bearer token for secure access.

- **Example snippet from sendListMessage():**

```
await axios.post(
  `https://graph.facebook.com/v22.0/${phoneNumberId}/messages`,
  {
    messaging_product: "whatsapp",
    to: recipient,
    type: "interactive",
    interactive: {
      type: "list",
      body: { text: message },
      footer: { text: "Team BLUEBEX" },
      action: {
        button: "Explore Our Services",
        sections: [
          { title: "Our Offerings", rows: [...] }
        ]
      }
    }
  },
  {
    headers: {
      "Content-Type": "application/json",
      Authorization: `Bearer ${config.whatsappToken}`,
    }
  }
);
```

2. Storing Outgoing Interactive Messages

- After successfully sending the message, each function calls the **storage service** to save the message in Firestore for future tracking and audit.
- This is done using the **ChatService.storeChatMessage()** function with fields:
    - **content**: The actual message text or body sent
    - **profileName**: The sender identity, e.g., "Bluebex"
    - **phoneNumber**: Recipient's WhatsApp number
    - **messageType**: Set as "outgoing_list" or "outgoing_button" depending on message type
    - **direction**: Always "outgoing" for messages sent by your system

- **Example from sendListMessage():**

```
await ChatService.storeChatMessage({
  content: message,
  profileName,
  phoneNumber: recipient,
  messageType: "outgoing_list",
  direction: "outgoing",
});
```

3. InteractiveHandlers.js

- **Purpose:** Centralizes the logic to handle and store all outgoing interactive messages consistently.
- It receives the outgoing message data and calls the storeChatMessage() function with correct parameters to keep a record in Firestore.
- It ensures **uniform metadata tagging** for all interactive outgoing messages, enabling easy filtering and querying (e.g., all button messages, or all list messages sent).

🗂️ Data Stored in Firestore (for Outgoing Interactive Messages)

Field	Description
content	The message body text or interactive message summary sent to the user
profileName	Sender identity, e.g., `"Bluebex"`
phoneNumber	Recipient's WhatsApp phone number
messageType	`"outgoing_list"` or `"outgoing_button"` to specify message kind
direction	`"outgoing"` indicates this message was sent by the system

TemplateHandler.js

Overview

templateHandler.js provides functions to send predefined WhatsApp **template messages** (using Meta's WhatsApp Cloud API) for different use cases — like requesting a quote or internship form submission. It also logs outgoing messages to Firebase via the ChatService.

This module exports two main async functions:

- sendQuoteForm — sends a quote request template message.
- sendInternshipQuoteForm — sends an internship request template message.

Both functions handle API communication, error logging, and saving message metadata.

Dependencies

- **axios**: for making HTTP POST requests to the WhatsApp Cloud API.
- **config**: contains environment variables, especially the WhatsApp API token.
- **logger**: custom logger utility for logging success and error messages.
- **ChatService**: service to store chat messages in Firebase or any configured DB.

Functions

1. sendQuoteForm(phoneNumberId, recipient, profileName = "Bluebex")

Sends a **quote request template message** to the specified WhatsApp recipient.

Parameters:

- phoneNumberId (string): Your WhatsApp Business phone number ID (used in the API URL).
- recipient (string): Recipient's WhatsApp phone number (in international format).
- profileName (string, optional): Display name for the profile in logs (default: "Bluebex").

Behavior:

- Constructs the API request URL with the phone number ID.
- Builds the message payload with the template name quote_request_info.

- Sends the template message via WhatsApp Cloud API.
- Logs success or error using logger.
- Stores a custom description of the message content in Firebase using ChatService.storeChatMessage.

---

2. sendInternshipQuoteForm(phoneNumberId, recipient, profileName = "Bluebex")

Sends an **internship form request template message** to the specified WhatsApp recipient.

Parameters:

- phoneNumberId (string): WhatsApp Business phone number ID.
- recipient (string): Recipient's WhatsApp phone number (international format).
- profileName (string, optional): Profile display name for logging (default: "Bluebex").

Behavior:

- Builds API URL and template message payload with template name internship_form.
- Sends the template message to the recipient.
- Logs success or failure.
- Saves a descriptive message in Firebase via ChatService.

Example usage:

```
const { sendQuoteForm, sendInternshipQuoteForm } = require("./templateHandler");

const phoneNumberId = "1234567890";
const recipient = "919876543210"; // recipient's phone number

// Send quote request template message
await sendQuoteForm(phoneNumberId, recipient);

// Send internship quote form template message
await sendInternshipQuoteForm(phoneNumberId, recipient);
```

FCM Notifications Handling Documentation

Overview

The application uses **Firebase Cloud Messaging (FCM)** to send push notifications to users, alerting them of new incoming messages. Each user is subscribed to a unique FCM topic based on their phone number, allowing precise and efficient delivery of notifications for their chats.

How Notifications Are Handled

- **Topic Naming:**
  The user's phone number (including country code) is used as the notification topic after removing the leading '+' sign. This creates a consistent topic ID for sending messages.
- **Notification Payload:**
  The notification includes:
  - A **title** (the sender's profile name or default "New Message").
  - A **body** containing the message content, truncated to 100 characters to avoid overly long notifications.
  - **Custom data** fields such as the formatted phone number and profile name for additional context.
- **Fallback Message:**
  If the message content is empty or contains only whitespace, a default notification body — "You have received a new message" — is used.
- **Sending Process:**
  Notifications are sent asynchronously using the Firebase Admin SDK's messaging().send() method.
- **Logging:**
  Both successful deliveries and errors are logged for monitoring and debugging purposes.

Function: `sendNotification(phoneNumber, messageContent, profileName)`

Parameter	Type	Description
`phoneNumber`	String	Recipient's phone number with country code (e.g., "+1234567890")
`messageContent`	String	Text content of the incoming message to display in the notification
`profileName`	String	Name of the sender/contact (optional)

Description

This function:

- Formats the phone number into a topic ID by stripping the '+' sign.
- Truncates the message content if longer than 100 characters.
- Provides a fallback notification body if the content is empty or blank.
- Builds the notification payload with the title, body, and additional data.
- Sends the notification using Firebase Admin SDK.
- Logs the result or any errors encountered.

Code Example

```
static async sendNotification(phoneNumber, messageContent, profileName) {
  try {
    // Remove '+' from phone number to form topic
    const formattedPhoneNumber = phoneNumber.replace('+', '');
    const channelID = formattedPhoneNumber;

    // Truncate message to 100 characters
    let notificationBody = messageContent;
    if (notificationBody.length > 100) {
      notificationBody = notificationBody.substring(0, 97) + "...";
    }

    // Use default message if content is empty or whitespace
    if (!notificationBody || notificationBody.trim() === "") {
      notificationBody = "You have received a new message";
    }

    // Prepare FCM payload
    const fcmMessage = {
      topic: channelID,
      data: {
        phoneNumber: formattedPhoneNumber,
        profileName: profileName || "User"
      },
      notification: {
        title: profileName || "New Message",
        body: notificationBody
      }
    };

    // Send notification via Firebase Admin SDK
    const response = await admin.messaging().send(fcmMessage);
    logger.info(`Notification sent to topic ${channelID}`, response);
    return response;

  } catch (error) {
    logger.warn(`No notification sent to ${phoneNumber}`, error.message);
    return null;
  }
}
```

Logging

- **Success:** Logs an info message with the topic ID and Firebase response.
- **Failure:** Logs a warning with the recipient's phone number and the error message.

Meta Application Version 2.0

Features Implementation with Backend Code

---

1. Message Sorting by Recent Messages (Recent on Top)

Feature:
Sort chat conversations so the most recent messages appear at the top.

Backend Implementation:

- Each chat has a lastMessage object updated on every new message.
- This object contains message content, timestamps, and date/time info.
- Frontend uses this to sort chats.

Code Snippet:

```
const timestamp = new Date().toISOString();
const date = new Date().toLocaleDateString('en-IN');  // e.g. "23/5/2025"
const time = new Date().toLocaleTimeString('en-US', { hour12: true, hour: '2-digit', minute: '2-d
const unixTimestamp = Math.floor(new Date().getTime() / 1000);

await db.ref(`chats/${message.phoneNumber}/lastMessage`).set({
  content: message.content,
  timestamp: timestamp,
  date: date,
  time: time,
  unixTimestamp: unixTimestamp,
});
```

2. Date and Time Format for Messages

Feature:
Show message timestamps as "Today", "Yesterday", or full date + time in readable format.

Backend Implementation:

- When saving messages, backend calculates and saves formatted date/time fields.
- These are stored inside each message to be displayed in UI.

Code Snippet:

```
const messageObj = {
  content: message.content,
  date: date, // formatted date string
  time: time, // formatted time string
  unixTimestamp: unixTimestamp,
  // other message metadata...
};

await db.ref(`chats/${message.phoneNumber}/messages/${message.messageId}`).set(messageObj);
```

3. "+ Button" for Sending Messages to New Numbers

Feature:
Allow sending a template message to a new phone number to initiate a chat (user consent required).

Backend Implementation:

- Created an API to send a Meta-approved template message via WhatsApp Cloud API.
- The template asks the user to reply "Hi" to start conversation.
- Application triggers this API with the target phone number and access token.

Code Snippet (triggering template message):

```
const sendTemplateMessage = async (phoneNumber) => {
  const response = await axios.post(`https://graph.facebook.com/v16.0/${WHATSAPP_PHONE_NUMBER_ID}/
    messaging_product: "whatsapp",
    to: phoneNumber,
    type: "template",
    template: {
      name: "your_template_name",
      language: { code: "en_US" }
    }
  }, {
    headers: {
      Authorization: `Bearer ${ACCESS_TOKEN}`
    }
  });
  return response.data;
};
```

4. Search Option in Chat List

Feature:
Users can search chats by message text, contact name, or phone number.

Implementation:

- Entirely frontend feature; no backend code required.

📬 5. WhatsApp Message Status & Unread Count Handling

🔹 Purpose

This function tracks WhatsApp message status updates (sent, delivered, read) and ensures that **unread message counts** in Firebase are accurate. It's essential for real-time chat sync and improving user experience with **read receipts** and **unread badge indicators**.

🔹 Role in the System

- Keeps chat conversations **in sync with WhatsApp read status**.
- Automatically **resets unreadCount** when a user reads a message.
- Supports **notification logic**, **badge counts**, and future **analytics** (e.g., delivery/read rate tracking).

🔹 **Code Logic**

```
const statuses = changes?.statuses;
if (statuses && statuses.length > 0) {
  const statusInfo = statuses[0];
  const status = statusInfo.status;
  const recipientId = statusInfo.recipient_id;

  logger.info(`📬 Status update received: ${status} for ${recipientId}`);

  if (status === "read") {
    await db.ref(`chats/${recipientId}/unreadCount`).set(0);
    logger.info(`✅ Unread count reset for ${recipientId}`);
  }

  return res.sendStatus(200);
}
```

🔹 **Firebase Structure (Simplified)**

```
chats/
  {recipientId}/
    unreadCount: 0
```

5a. Unread Message Count per Chat

Feature:
Show unread message count next to each chat in chat list.

Backend Implementation:

- Increment unreadCount in Firebase when an incoming message arrives.

Code Snippet:

```
if (messageData.direction === "incoming") {
  await db.ref(`chats/${message.phoneNumber}/unreadCount`).transaction((current) => {
    return (parseInt(current) || 0) + 1;
  });
}
```

5b. Unread Conversations Count (Global)

Feature:
Maintain count of how many chats have unread messages.

Backend Implementation:

- When unreadCount for a chat changes from 0 to 1, increment the global unreadConversations count.

Code Snippet:

```
async function updateUnreadConversations(phoneNumber) {
  const unreadCountRef = db.ref(`chats/${phoneNumber}/unreadCount`);
  const unreadConversationsRef = db.ref('chats/unreadConversations');

  const snapshot = await unreadCountRef.once('value');
  const unreadCount = parseInt(snapshot.val() || '0');

  if (unreadCount === 1) {
    await unreadConversationsRef.transaction(current => (parseInt(current) || 0) + 1);
  }
}
```

This function should be called after incrementing the unread count for each incoming message.

Summary

Feature	Backend Code Location	Key Functions/References
Sort messages by recent	`lastMessage` update	`db.ref(.../lastMessage).set()`
Date/Time formatting	Message store function	`date`, `time`, `unixTimestamp` stored per message
"+ Button" new user message	Template message API call	Axios POST to WhatsApp Cloud API with template payload
Search in chat list	Frontend only	No backend changes required
Unread message count	`unreadCount` transaction	`db.ref(.../unreadCount).transaction()`
Unread conversations count	Global unreadConversations	`updateUnreadConversations()` function

📦 Firebase Chat Storage Migration Documentation

✅ Overview

This update restructures how chat data is stored in Firebase Realtime Database. Previously, messages were stored using phoneNumber as the primary key. We've now **migrated to a userId-centric structure** to improve scalability, indexing, and data consistency.

🔁 What Changed?

🆕 New Data Path Structure

Before:

```
/chats/{phoneNumber}/messages
```

After:

```
/chats/{userId}/{phoneNumber}/messages
```

📁 Updated Database Paths

Feature	Old Path	New Path
Store Raw Form Submission	`/chats/{phoneNumber}/messages`	`/chats/{userId}/{phoneNumber}/messages`
Store Profile Info	`/chats/{phoneNumber}/profileName` `/phoneNumber`	`/chats/{userId}/{phoneNumber}/profileName` `/phoneNumber`
Store Formatted Messages	`/chats/{phoneNumber}/messages`	`/chats/{userId}/{phoneNumber}/messages`
Store Last Message	`/chats/{phoneNumber}/lastMessage`	`/chats/{userId}/{phoneNumber}/lastMessage`
Update Unread Count	`/chats/{phoneNumber}/unreadCount`	`/chats/{userId}/{phoneNumber}/unreadCount`
Track Unread Conversations	`/chats/{phoneNumber}/unreadCount`	`/chats/{userId}/{phoneNumber}/unreadCount` + `/chats/unreadConversations`

🧠 User ID Mapping Logic

Path Used for Mapping:

```
/userIds/{phoneNumber} → userId
```

Mapping Behavior:

- On first contact, a new userId is **generated** and saved.
- On repeat contact, the **same userId is reused** (no duplicates).

📂 Folder Structure Snapshot

swift Copy Edit

```
/userIds/{phoneNumber} = userId
/chats/{userId}/{phoneNumber}/
  ├── profileName
  ├── phoneNumber
  ├── messages/
  ├── lastMessage
  └── unreadCount
/chats/unreadConversations
```

🔧 Updated Code

🔸 UserIdService.js

```
const { db } = require("../config/firebase");

class UserIdService {
  static generateUserId() {
    const randomSixDigits = Math.floor(100000 + Math.random() * 900000);
    const userId = `USERID${randomSixDigits}`;
    console.log(`[UserIdService] Generated userId: ${userId}`);
    return userId;
  }

  static async getOrCreateUserId(phoneNumber) {
    const userIdRef = db.ref(`/userIds/${phoneNumber}`);
    const snapshot = await userIdRef.once('value');
    let userId = snapshot.val();

    if (userId) {
      console.log(`[UserIdService] Found existing userId for ${phoneNumber}: ${userId}`);
      return userId;
    } else {
      userId = this.generateUserId();
      await userIdRef.set(userId);
      console.log(`[UserIdService] Stored new userId for ${phoneNumber}: ${userId}`);
      return userId;
    }
  }
}

module.exports = UserIdService;
```

🔸 ChatService.js

```
const { db } = require("../config/firebase");
const UserIdService = require("./UserIdService");

class ChatService {
  static async storeChatMessage(message) {
    try {
      const userId = await UserIdService.getOrCreateUserId(message.phoneNumber);
      const timestamp = message.timestamp || new Date().toISOString();

      if (message.messageType === "form_submission") {
        logger.info(`Storing form submission for ${message.phoneNumber}`);
        await db.ref(`chats/${userId}/${message.phoneNumber}/messages`).push(message);

        if (message.direction === "incoming") {
          await this.sendNotification(message.phoneNumber, message.content || "New form submissior
        }
        return;
      }

      if (message.messageType === "interactive") {
        message.content = `User clicked on '${message.content}' button`;
      }

      const dateObj = new Date(timestamp);
      const date = dateObj.toISOString().split("T")[0];
      const time = dateObj.toLocaleTimeString("en-IN", { hour: '2-digit', minute: '2-digit', hour1
      const unixTimestamp = Math.floor(dateObj.getTime() / 1000);
```

Copy Edit

🧪 **Testing & Validation**

After changes, tested:

- ✅ Storing new messages
- ✅ Updating last message
- ✅ Correct unreadCount increment
- ✅ Accurate userId mapping

- ✅ Notification dispatch

➢ **Contributions:**

- Balaji and Manikanta

- **Research and Development of WhatsApp APIs from Basics to Advanced**

  It Includes:

  1. Meta Developer Account Setup
  2. Generating Permanent Access Token
  3. Creating WhatsApp Templates
  4. Understanding Meta Policies

     4a. 24 Hours customer Service Window

     4b. Business Initiation messages without User consent using pre-approved meta-Templates

- **WhatsApp Cloud API- Different Type of Messages**

  It Includes:

  1. Text Messages
  2. Template Message
  3. Image with Message
  4. Document Message
  5. Audio Message
  6. Contact Message
  7. Interactive Message
  8. Getting the Media Id By sending the Image or document
  9. Location Message
  10.

- **Advanced work on WhatsApp API integration and Chatbot Development**

  It Includes:

  - Webhook Setup
  - Automated Replies
  - Interactive Chatbot
  - Using Media Handling
  - WhatsApp And Chatbot integration

- T**eam Collaboration — WhatsApp Receipt Integration with APIs in Mallikarjuna's project! (Document API)**

1.Provided API for Sending Receipt dynamically by using Document API which automatically sends Receipts to the customer WhatsApp Mobile Number.

- **Chatbot Foundation Setup**

  It includes:

  1. Simple Automated Chatbot with no interactive messages
  2. Profile Name Extraction
  3. Webhook Integration
  4. Enhancing the Chatbot with Interactive messages with Services like
     1. Web development
     2. App development
     3. DevOps
     4. AI&ML Services

     Service Selection Via Interactive Menu buttons
  5. Deployed on Render.com

- META Webhook Configuration in Meta Account

  Key Components:

  1. META Developers Account Creation
  2. Phone Number ID
  3. Permanent Access Token
  4. Webhook Callback URL along with verify token
  5. Webhook Verification

- **Manikanta Vaddi (WhatsApp Chatbot Backend)**
  - Template Handling (Internship template)
  - Firebase Admin SDK with Node.js
  - Initial Setup with Realtime Firebase
  - Setup environmental variables
  - Firebase Admin initialisation
  - Sending a list message
  - Interactive messages (Software Development)
    It includes
    - App Development

    - Web development
    - Saas Products

- Interactive Menu (internships & Courses)
  It includes
    - Android Development
    - Web Development
    - IOS Development
- Storing the incoming messages in Firebase
  It includes
    - Messaging Type: Text
    - Messaging Type: Interactive
- Storing the outgoing messages in Firebase
  It includes
    - Messaging Type: outgoing list
    - Messaging Type: outgoing template
- R&D on Triggering a FCM Notification
- Sort the message according to the recent messages (Recent on Top)
    - Implemented backend field as last Message in firebase because of which sorting the messages on top in the application was successfully implemented.
- + Button (Add button) in the chat screen to be assigned to send a message to a new number
    - Since we can't send directly without user consent for a new number, we have created a Meta-approved template to trigger the message using a template, saying the user to send "Hi" to this number in order to get started with the conversation.
- Unread Message count in the application for the unread messages in chat list Implemented.
    - We implemented an unread message counter that increments whenever a new incoming message is received and stored in Firebase under the specific user's chat node.
    - This counter is updated using Firebase transactions, ensuring atomic operations that prevent data corruption or race conditions—especially when multiple messages arrive simultaneously.
    - The unread count is stored at: chats/{phoneNumber}/unreadCount & is incremented for every incoming message
- Migrated Chat Message Storage to a User-Centric Structure

- Successfully migrated chat storage to use userId as the primary key instead of phoneNumber directly.
- New message storage path: /chats/{userId}/{phoneNumber}/messages

4 Storing Last Message

Before: chats/${message.phoneNumber}/lastMessage

After: chats/${userId}/${message.phoneNumber}/lastMessage

5 Updating Unread Count

Before: chats/${message.phoneNumber}/unreadCount

After: chats/${userId}/${message.phoneNumber}/unreadCount

6 Helper Function: updateUnreadConversations()

Before: chats/${phoneNumber}/unreadCount

After: chats/${userId}/${phoneNumber}/unreadCount

- **Balaji S (WhatsApp Chatbot Backend)**
    - Webhook Verification
    - Extracting Profile Name from Webhooks
    - Template Handling (Business template)
    - Processing Webhook Controller
    - Interactive menu (Automation Tools)
      It includes
        - Invoicing and Estimate
        - Custom ERP solutions
        - WhatsApp Integration
        - Inventory Management
    - Talk to a Human
      It includes
        - Sending a Custom Message
        - Product Demo message
        - Sending a Contact us message

    - Storing the incoming messages in Firebase
      It includes
        - Messaging Type: Form submission
        - Messaging Type: Template
    - Storing the Outgoing messages in Firebase
      It includes
        - Messaging Type: outgoing text
        - Messaging Type: outgoing Button
        - Messaging Type: Form submission
    - Triggering a FCM Notification
        - For incoming Messages
    - Date format for message as in WhatsApp. (Today, Yesterday, Friday, Thursday, Wednesday, Tuesday
        - Backend work: We have created a lastMessage object in firebase inside that, date, time, along with timestamp and Unix timestamp added so that the date and time format is visible

- + Button (Add button) in the chat screen to be assigned to send a message to a new number
    - We have shared the API for triggering a template message by using a phone number.
    - Application team used that API with access token to trigger template message for new users using a phonenumber
- Unread conversations in the application for the chats which are unread types of conversations is successfully implemented
    - We implemented an unread conversations counter to track how many unique chats currently contain one or more unread messages.
    - This count is stored centrally at: chats/unreadConversations
    - Each time an incoming message is received and the chat's unreadCount becomes 1, the backend logic assumes that the conversation has just become unread.
    - In such cases, the global unreadConversations count is incremented by 1 using a Firebase transaction
- Migrated Chat Message Storage to a User-Centric Structure
    - Created a Function for Generating a user id
    - Successfully migrated chat storage to use userId as the primary key instead of phoneNumber directly.
    - New message storage path: /chats/{userId}/{phoneNumber}/messages

➢ Updated Firebase Database Paths:
1 Storing Raw Form Submission

Before: chats/${message.phoneNumber}/messages

After: chats/${userId}/${message.phoneNumber}/messages

2 Storing Profile Info

Before:

chats/${message.phoneNumber}/profileName

chats/${message.phoneNumber}/phoneNumber

After:

chats/${userId}/${message.phoneNumber}/profileName

chats/${userId}/${message.phoneNumber}/phoneNumber

3 Storing Formatted Message

Before: chats/${message.phoneNumber}/messages

After: chats/${userId}/${message.phoneNumber}/messages

----------END OF REPORT----------