Backend Script

```
from flask import Flask, request, jsonify, render_template, abort
from flask_cors import CORS
from groq_test import FinanceBot
import logging
import os

# Initialize Flask app with template folder
app = Flask(__name__, template_folder='templates')

CORS(app, origins=[
    "http://localhost:*",
    "http://127.0.0.1:*",
    "https://nitram-db-finance-bot-llm-1.onrender.com"
], supports_credentials=True)

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Initialize FinanceBot
try:
    bot = FinanceBot()
    logger.info("FinanceBot initialized successfully")

    # Use the new test_connection() method here
    if not bot.test_connection():
        raise RuntimeError("Database test query failed")
    logger.info("▫ FinanceBot connection verified")

except Exception as e:
    logger.error(f"Failed to initialize FinanceBot: {str(e)}")
    raise

# Serve HTML from templates folder
@app.route('/')
def serve_index():
    try:
        return render_template('index.html')
    except Exception as e:
        logger.error(f"Error loading index.html: {str(e)}")
        abort(404, description="index.html not found in templates folder.")

# API endpoint for chat queries
@app.route('/ask', methods=['POST'])
def ask():
    try:
        logger.info(f"Incoming request headers: {request.headers}")
        logger.info(f"Request data: {request.data}")

        if not request.is_json:
            logger.warning("Request missing JSON content type")
            return jsonify({'error': 'Content-Type must be application/json'}), 415

        data = request.get_json()

        if not data or 'query' not in data:
            logger.warning("Missing query parameter")
```

```
            return jsonify({'error': 'Missing query parameter'}), 400

        user_query = data['query'].strip()
        if not user_query:
            logger.warning("Empty query received")
            return jsonify({'error': 'Query cannot be empty'}), 400

        logger.info(f"Processing query: {user_query}")

        # Process query through FinanceBot
        bot_response = bot.ask(user_query)
        logger.info(f"Generated response: {bot_response[:200]}...")

        return jsonify({
            'response': bot_response,
            'status': 'success'
        })

    except Exception as e:
        logger.error(f"Error processing request: {str(e)}", exc_info=True)
        return jsonify({
            'error': 'An error occurred while processing your request',
            'status': 'error'
        }), 500

if __name__ == '__main__':
    app.run(host='127.0.0.1', port=5000, debug=True)

Database + LLM Groq API Intergaration Script

import os
import re
import json
import logging
import requests
import psycopg2
from psycopg2 import pool
from urllib.parse import urlparse
from dotenv import load_dotenv
from datetime import datetime, date
from decimal import Decimal

# Load environment variables
load_dotenv()

# Set up logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

class FinanceBot:
    def __init__(self):
        # connect DATABASE_URL
        db_url = os.getenv("DATABASE_URL")
        parsed = urlparse(db_url)

        self.db_pool = psycopg2.pool.SimpleConnectionPool(
            minconn=1,
            maxconn=10,
            user=parsed.username,
```

```
            password=parsed.password,
            host=parsed.hostname,
            port=parsed.port,
            database=parsed.path[1:]
        )

        self.schema = self._load_schema()
        self.context = {
            'current_account': None,
            'current_customer': None,
            'last_query_type': None,
            'conversation_history': []
        }

    def test_connection(self):
        conn = self.db_pool.getconn()
        try:
            with conn.cursor() as cursor:
                cursor.execute("SELECT 1")
                return cursor.fetchone()[0] == 1
        except Exception as e:
            logger.error(f"Database connection test failed: {str(e)}")
            return False
        finally:
            self.db_pool.putconn(conn)

    def _load_schema(self):
        conn = self.db_pool.getconn()
        try:
            with conn.cursor() as cursor:
                cursor.execute("""
                    SELECT table_name, column_name, data_type
                    FROM information_schema.columns
                    WHERE table_schema = 'public'
                    ORDER BY table_name, ordinal_position;
                """)
                schema = {}
                for table, column, dtype in cursor.fetchall():
                    schema.setdefault(table, []).append((column, dtype))
                return schema
        except Exception as e:
            logger.error(f"Error loading schema: {str(e)}")
            return {}
        finally:
            self.db_pool.putconn(conn)

    def _generate_sql(self, user_query):
        if not self.schema:
            return None
        try:
            schema_info = "\n".join(
                f"Table {table}: {', '.join(f'{col} ({dtype})' for col, dtype in
columns)}"
                for table, columns in self.schema.items()
            )
            history = "\n".join([f"User: {q}\nBot: {a}" for q, a in
self.context['conversation_history'][-3:]])

            response = requests.post(
```

```
                "https://api.groq.com/openai/v1/chat/completions",
                headers={
                    "Authorization": f"Bearer {os.getenv('GROQ_API_KEY')}",
                    "Content-Type": "application/json"
                },
                json={
                    "model": os.getenv("GROQ_MODEL", "llama3-8b-8192"),
                    "messages": [
                        {"role": "system", "content": f"""
                            You are a financial SQL expert. Convert user questions
into PostgreSQL queries.
                            Database Schema:
                            {schema_info}
                            Conversation History:
                            {history}
                            Context:
                            {json.dumps(self.context)}
                            Rules:
                            1. Use lowercase table/column names.
                            2. Join tables where necessary.
                            3. For totals, use SUM().
                            4. Return SQL inside ```sql``` blocks only.
                        """},
                        {"role": "user", "content": user_query}
                    ],
                    "temperature": 0.3,
                    "max_tokens": 500
                },
                timeout=30
            )

            response.raise_for_status()
            content = response.json()['choices'][0]['message']['content']
            match = re.search(r"```sql\s*(.*?)```", content, re.DOTALL |
re.IGNORECASE)
            return match.group(1).strip() if match else None
        except Exception as e:
            logger.error(f"SQL generation error: {str(e)}")
            return None

    def _execute_query(self, sql):
        conn = self.db_pool.getconn()
        try:
            with conn.cursor() as cursor:
                cursor.execute(sql)
                rows = cursor.fetchall()
                headers = [desc[0] for desc in cursor.description]
                return [dict(zip(headers, row)) for row in rows]
        except Exception as e:
            logger.error(f"Query execution error: {str(e)}")
            return None
        finally:
            self.db_pool.putconn(conn)

    def _generate_natural_response(self, data, user_query):
        if not data:
            return "Sorry, I couldn’t find any matching records."

        try:
```

```
            def format_value(v):
                if isinstance(v, (Decimal, float, int)):
                    return float(v)
                if isinstance(v, (datetime, date)):
                    return v.strftime("%Y-%m-%d")
                return str(v)

            formatted_data = [
                {k: format_value(v) for k, v in row.items()} for row in data
            ]

            response = requests.post(
                "https://api.groq.com/openai/v1/chat/completions",
                headers={
                    "Authorization": f"Bearer {os.getenv('GROQ_API_KEY')}",
                    "Content-Type": "application/json"
                },
                json={
                    "model": os.getenv("GROQ_MODEL", "llama3-8b-8192"),
                    "messages": [
                        {"role": "system", "content": """
                            You are a professional financial assistant for a bank.
Given a user query and matching database records,
                            generate a clear, accurate, and concise response. Your
tone should reflect how a bank staff member communicates
                            in a professional setting—whether summarizing data for
internal review or preparing information to relay
                            to a customer. Focus on key figures, avoid unnecessary
filler, and vary your phrasing naturally based on
                            the query type.
                        """},
                        {"role": "user", "content": f"""
                            User asked: \"{user_query}\"

                            Here is the data from the database that matches the
query:
                            {json.dumps(formatted_data, indent=2)}
                        """}
                    ],
                    "temperature": 0.7,
                    "max_tokens": 300
                },
                timeout=30
            )

            response.raise_for_status()
            reply = response.json()['choices'][0]['message']['content'].strip()
            return reply
        except Exception as e:
            logger.error(f"Natural response generation error: {str(e)}")
            return f"{len(data)} records found, but I couldn't format a response."

    def _update_context(self, user_query, response_text):
        self.context['conversation_history'].append((user_query, response_text))
        if len(self.context['conversation_history']) > 5:
            self.context['conversation_history'] =
self.context['conversation_history'][-5:]

    def ask(self, user_query):
```

```
        sql = self._generate_sql(user_query)
        if not sql:
            return "Sorry, I couldn't generate a valid SQL query for that."
        results = self._execute_query(sql)
        if results is None:
            return "Sorry, I couldn't retrieve any data for that."
        response = self._generate_natural_response(results, user_query)
        self._update_context(user_query, response)
        return response

# CLI Entry point
if __name__ == "__main__":
    try:
        bot = FinanceBot()
        print("Nitram BankBot\nHello! I'm your Nitram Bank assistant. How can I
help you today?")
        while True:
            try:
                user_input = input("You: ").strip()
                if user_input.lower() in ['exit', 'quit']:
                    print("Nitram: Alright, talk to you later.")
                    break
                reply = bot.ask(user_input)
                print(f"Nitram: {reply}")
            except KeyboardInterrupt:
                print("\nNitram: Session ended. Take care!")
                break
    except Exception as e:
        logger.error(f"Bot failed to start: {str(e)}")
```

Frontend Script

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
  <title>Nitram Cooperative BankBot Chat</title>
  <style>
    body {
      margin: 0;
      font-family: Arial, sans-serif;
      display: flex;
      height: 100vh;
      overflow: hidden;
    }
    .sidebar {
      width: 290px;
      background-color: #f5f5f5;
      border-right: 1px solid #ddd;
      display: flex;
      flex-direction: column;
      padding: 16px;
    }
    .sidebar h2 {
      margin-top: 0;
    }
    .sidebar .disclaimer {
```

```
  font-size: 12px;
  font-style: italic;
  color: #666;
  margin: 5px 0 15px 0;
  line-height: 1.3;
}
.sidebar button {
  background-color: #004aad;
  color: white;
  border: none;
  padding: 10px 16px;
  border-radius: 6px;
  cursor: pointer;
}
.sidebar button:hover {
  background-color: #00307d;
}
.history, .insights {
  flex: 1;
  overflow-y: auto;
  margin-top: 20px;
}
.chat-container {
  flex: 1;
  display: flex;
  flex-direction: column;
}
.chat-header {
  background-color: #004aad;
  color: white;
  padding: 12px;
  display: flex;
  justify-content: space-between;
  align-items: center;
}
.chat-box {
  flex: 1;
  padding: 20px;
  overflow-y: auto;
  background: #eef1f5;
}
.chat-input {
  padding: 16px;
  border-top: 1px solid #ccc;
  background-color: #fff;
  display: flex;
  flex-direction: column;
  gap: 10px;
}
.chat-input-row {
  display: flex;
  gap: 10px;
}
.chat-input textarea {
  flex: 1;
  padding: 10px;
  border-radius: 8px;
  resize: none;
  border: 1px solid #ccc;
```

```
}
.chat-input button {
  background-color: #004aad;
  color: white;
  border: none;
  padding: 10px 16px;
  border-radius: 6px;
  cursor: pointer;
}
.chat-input button:hover {
  background-color: #00307d;
}
.message {
  margin-bottom: 16px;
}
.message.user {
  text-align: right;
}
.message.bot {
  text-align: left;
}
.message-content {
  display: inline-block;
  padding: 10px;
  border-radius: 10px;
  max-width: 70%;
}
.user .message-content {
  background-color: #d1e8ff;
}
.bot .message-content {
  background-color: #fff;
  border: 1px solid #ddd;
}
.bot .message-card {
  background: white;
  border-radius: 10px;
  box-shadow: 0 2px 8px rgba(0,0,0,0.1);
  padding: 12px 16px;
  max-width: 80%;
  margin-bottom: 12px;
}
.message-card h3 {
  margin-top: 0;
  color: #004aad;
  font-size: 16px;
}
.message-card p {
  margin: 8px 0;
}
.message-card ul {
  padding-left: 20px;
}
.message-card li {
  margin-bottom: 6px;
}
.typing-indicator {
  display: inline-flex;
  padding: 10px 15px;
```

```
      background: white;
      border-radius: 18px;
      box-shadow: 0 2px 8px rgba(0,0,0,0.1);
    }
    .typing-dot {
      width: 8px;
      height: 8px;
      background: #666;
      border-radius: 50%;
      margin: 0 3px;
      animation: typingAnimation 1.4s infinite ease-in-out;
    }
    .typing-dot:nth-child(2) {
      animation-delay: 0.2s;
    }
    .typing-dot:nth-child(3) {
      animation-delay: 0.4s;
    }
    @keyframes typingAnimation {
      0%, 60%, 100% { transform: translateY(0); }
      30% { transform: translateY(-4px); }
    }
    .history-item {
      padding: 8px;
      border-bottom: 1px solid #ccc;
      cursor: pointer;
    }
    .history-item:hover {
      background-color: #e0e0e0;
    }
    .insight-item {
      padding: 6px 0;
      font-size: 14px;
      color: #333;
    }
  </style>
</head>
<body>
  <div class="sidebar">
    <h2>Nitram Cooperative Bank</h2>
    <div class="disclaimer">For authorized bank officials and administrators
only</div>
    <button onclick="newChat()">New Chat</button>
    <div class="history" id="history">
      <h3>Searches</h3>
    </div>
    <div class="insights">
      <h3>Insights</h3>
      <div class="insight-item">▫ How to budget</div>
      <div class="insight-item">▫ Understanding loan terms</div>
      <div class="insight-item">▫ Avoiding debt traps</div>
      <div class="insight-item">▫ Gold Savings Plan explained</div>
      <div class="insight-item">▫ Improve your credit score</div>
    </div>
  </div>

  <div class="chat-container">
    <div class="chat-header">
      <div id="chat-title">Admin BankBot</div>
```

```
    </div>
    <div class="chat-box" id="chat-box">
      <!-- Messages will appear here -->
    </div>
    <div class="chat-input">
      <button onclick="newChat()" style="align-self: flex-start;">New Chat</button>
      <div class="chat-input-row">
        <textarea id="chat-input" rows="1" placeholder="Type a message...
😊"></textarea>
        <button onclick="sendMessage()">Send</button>
      </div>
    </div>
  </div>

  <script>
    let chatHistory = [];
    let currentChat = [];
    let isBotTyping = false;

    // Initialize with welcome message
    window.onload = function() {
      appendMessage("bot", "Hello! I'm your Nitram Bank assistant. How can I help
you today?");
      currentChat.push({ sender: "bot", text: "Hello! I'm your Nitram Bank
assistant. How can I help you today?" });
    };

    function newChat() {
      if (currentChat.length > 0) {
        if (chatHistory.length >= 10) chatHistory.shift();
        chatHistory.push([...currentChat]);
        updateHistory();
      }
      currentChat = [];
      document.getElementById("chat-box").innerHTML = "";
      document.getElementById("chat-title").textContent = "New Chat";
      appendMessage("bot", "Hello! I'm your Nitram Bank assistant. How can I help
you today?");
      currentChat.push({ sender: "bot", text: "Hello! I'm your Nitram Bank
assistant. How can I help you today?" });
    }

    function updateHistory() {
      const historyDiv = document.getElementById("history");
      const chats = historyDiv.querySelectorAll(".history-item");
      chats.forEach((c) => c.remove());
      chatHistory.forEach((chat, index) => {
        const el = document.createElement("div");
        el.className = "history-item";
        el.textContent = chat[0]?.text || `Chat ${index + 1}`;
        el.onclick = () => loadChat(index);
        historyDiv.appendChild(el);
      });
    }

    function loadChat(index) {
      const chat = chatHistory[index];
      const chatBox = document.getElementById("chat-box");
      chatBox.innerHTML = "";
```

```
      chat.forEach((msg) => {
        appendMessage(msg.sender, msg.text);
      });
      document.getElementById("chat-title").textContent = chat[0]?.text || "History
Chat";
    }

    function sendMessage() {
      if (isBotTyping) return;

      const input = document.getElementById("chat-input");
      const text = input.value.trim();
      if (!text) return;

      if (currentChat.length === 0) {
        document.getElementById("chat-title").textContent = text.length > 30 ? `$
{text.substring(0, 30)}...` : text;
      }

      appendMessage("user", text);
      currentChat.push({ sender: "user", text });
      input.value = "";

      showTypingIndicator();
      getBotResponse(text);
    }

    function showTypingIndicator() {
      isBotTyping = true;
      const typingDiv = document.createElement("div");
      typingDiv.className = "message bot";
      typingDiv.innerHTML = `
        <div class="typing-indicator">
          <div class="typing-dot"></div>
          <div class="typing-dot"></div>
          <div class="typing-dot"></div>
        </div>
      `;
      document.getElementById("chat-box").appendChild(typingDiv);
      document.getElementById("chat-box").scrollTop =
document.getElementById("chat-box").scrollHeight;
      return typingDiv;
    }

    function removeTypingIndicator() {
      const chatBox = document.getElementById("chat-box");
      const typingIndicators = chatBox.querySelectorAll('.typing-indicator');
      typingIndicators.forEach(indicator => {
        indicator.parentElement.remove();
      });
      isBotTyping = false;
    }

    async function getBotResponse(input) {
      const apiUrl = "https://nitram-db-finance-bot-llm-1.onrender.com/ask";
      const typingDiv = showTypingIndicator();

      try {
        console.log("[DEBUG] Sending query:", input);
```

```
    const response = await fetch(apiUrl, {
      method: "POST",
      mode: "cors",
      credentials: "include",
      headers: {
        "Content-Type": "application/json",
        "Accept": "application/json"
      },
      body: JSON.stringify({ query: input })
    });

    console.log("[DEBUG] Response status:", response.status);

    if (!response.ok) {
      const errorText = await response.text();
      console.error("[ERROR] Server response:", errorText);
      throw new Error(`Server responded with status ${response.status}`);
    }

    const data = await response.json();
    console.log("[DEBUG] Received data:", data);

    removeTypingIndicator();

    if (data && data.response) {
      await streamResponse(data.response);
      currentChat.push({ sender: "bot", text: data.response });
    } else {
      throw new Error("Invalid response format from server");
    }

  } catch (error) {
    console.error("[ERROR] Full error:", error);
    removeTypingIndicator();

    let errorMsg = "Could not connect to the server. Please:";
    errorMsg += "\n1. Ensure backend is running";
    errorMsg += "\n2. Check browser console for details";
    errorMsg += "\n3. Try refreshing the page";

    appendMessage("bot", errorMsg);
  }
}

async function streamResponse(text) {
  let messageContent;
  try {
    const data = JSON.parse(text);
    if (Array.isArray(data)) {
      messageContent = formatAsCard(data);
    } else {
      messageContent = text;
    }
  } catch {
    messageContent = text;
  }

  const div = document.createElement("div");
```

```
      div.className = "message bot";
      const content = document.createElement("div");
      content.className = typeof messageContent === 'string' ? "message-content" :
"message-card";
      div.appendChild(content);
      document.getElementById("chat-box").appendChild(div);

      const words = typeof messageContent === 'string'
        ? messageContent.split(' ')
        : [messageContent];

      for (let i = 0; i < words.length; i++) {
        if (typeof words[i] === 'string') {
          content.textContent += (i > 0 ? ' ' : '') + words[i];
        } else {
          content.innerHTML = words[i];
        }
        await new Promise(resolve => setTimeout(resolve, 50));
        document.getElementById("chat-box").scrollTop =
document.getElementById("chat-box").scrollHeight;
      }
    }

    function formatAsCard(data) {
      if (data.length === 0) return "<p>No data found</p>";

      let html = '';
      if ('balance' in data[0]) {
        html = `<h3>Account Balances</h3><ul>`;
        data.forEach(item => {
          html += `<li><strong>${item.account}:</strong> ${item.balance}</li>`;
        });
        html += `</ul>`;
      } else if ('amount' in data[0]) {
        html = `<h3>Recent Transactions</h3><ul>`;
        data.forEach(item => {
          html += `<li>${item.date}: ${item.description}
<strong>${item.amount}</strong></li>`;
        });
        html += `</ul>`;
      } else {
        html = `<h3>Information</h3><p>${JSON.stringify(data)}</p>`;
      }
      return html;
    }

    function appendMessage(sender, text) {
      const div = document.createElement("div");
      div.className = `message ${sender}`;

      try {
        const data = JSON.parse(text);
        if (Array.isArray(data)) {
          const card = document.createElement("div");
          card.className = "message-card";
          card.innerHTML = formatAsCard(data);
          div.appendChild(card);
        } else {
          const content = document.createElement("div");
```

```
          content.className = "message-content";
          content.textContent = text;
          div.appendChild(content);
        }
      } catch {
        const content = document.createElement("div");
        content.className = "message-content";
        content.textContent = text;
        div.appendChild(content);
      }

      const chatBox = document.getElementById("chat-box");
      chatBox.appendChild(div);
      chatBox.scrollTop = chatBox.scrollHeight;
    }

    document.getElementById('chat-input').addEventListener('keydown', function(e) {
      if (e.key === 'Enter' && !e.shiftKey) {
        e.preventDefault();
        sendMessage();
      }
    });
  </script>
</body>
</html>
```