Smart Shop v2.0: Technical Documentation

1. Project Overview

This document details the complete backend for the Smart Shop v2.0 application. The server is built in **Node.js** with the **Express** framework. It connects to a **Supabase (PostgreSQL)** database and uses the **Google Gemini API** for three advanced AI features: AI-driven product search, image-based food recognition, and market trend analysis.

2. Part 1: Project Setup & Configuration

These files set the foundation for your project, defining its dependencies and secret keys.

A. The `package.json` Dependencies

This section lists the "shopping list" of tools your project needs.

```
{
  "dependencies": {
    "@google/generative-ai": "^0.24.1",
    "@supabase/supabase-js": "^2.57.4",
    "dotenv": "^17.2.2",
    "express": "^5.1.0",
    "multer": "^2.0.2"
  }
}
```

`@google/generative-ai`: The official Google library that lets your server connect to and "talk" to the Gemini AI models.

`@supabase/supabase-js`: The official Supabase library. This is the "bridge" that lets your code easily query your database (e.g., `supabase.from().select()`) and handle user authentication.

`dotenv`: A critical utility that loads your secret keys from the `.env` file into your application so your code can use them.

`express`: The web server framework itself. This is the "engine" that creates your server, manages your API endpoints (URLs), and handles HTTP requests.

`multer`: A special helper for Express. Its only job is to handle file uploads, which you need for the "search by image" feature.

B. The `.env` Environment File

This file **must never be shared**. It stores your secrets. The `dotenv` package reads this file.

SUPABASE_URL=YOUR_SUPABASE_URL

SUPABASE_ANON_KEY=YOUR_SUPABASE_ANON_KEY

GEMINI_API_KEY=YOUR_GEMINI_API_KEY

`SUPABASE_URL`: The unique address for your database project.

`SUPABASE_ANON_KEY`: Your public "ID card" for the Supabase project.

`GEMINI_API_KEY`: Your private, secret password for all Google AI services.

3. Part 2: The Database (Complete SQL Setup)

This SQL script creates your entire database structure.

A. Table Creation

These four tables store all your data.

-- 1. Table for user profiles, linked to the built-in auth.users

CREATE TABLE public.profiles (

  id uuid REFERENCES auth.users NOT NULL PRIMARY KEY,

  full_name TEXT,

  email TEXT);

```
-- 2. Table for shops, linked to a profile
-- Includes new fields for contact info
CREATE TABLE public.shops (
  id BIGINT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  name TEXT NOT NULL,
  description TEXT,
  latitude NUMERIC,
  longitude NUMERIC,
  owner_id uuid NOT NULL REFERENCES public.profiles(id),
  contact_number TEXT,
  email TEXT);
-- 3. Table for products, linked to a shop
-- Includes a 'rating' for better AI sorting
CREATE TABLE public.products (
  id BIGINT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  name TEXT NOT NULL,
  description TEXT,
  price NUMERIC NOT NULL,
  rating NUMERIC DEFAULT 3.0,
  shop_id BIGINT NOT NULL REFERENCES public.shops(id));
```

```
-- 4. Table for Market Trend Analysis
CREATE TABLE public.market_price (
  id BIGINT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  created_at TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT now(),
  product_name TEXT NOT NULL,
  price NUMERIC NOT NULL,
  source TEXT
);
```

B. Automatic Profile Trigger

When a user signs up with Supabase Auth, they are added to the `auth.users` table. This code automatically runs, copying that new user's ID, name, and email into your "public" `profiles` table so you can link shops to it.

```
CREATE FUNCTION public.handle_new_user()
RETURNS TRIGGER
LANGUAGE plpgsql
SECURITY DEFINER SET search_path = public
AS $$
BEGIN
  INSERT INTO public.profiles (id, full_name, email)
  VALUES (new.id, new.raw_user_meta_data->>'full_name', new.email);
  RETURN new;
END;
$$;
```

```
CREATE TRIGGER on_auth_user_created
  AFTER INSERT ON auth.users
  FOR EACH ROW EXECUTE PROCEDURE public.handle_new_user();
```

C. Location Search Function

This SQL function uses the **PostGIS** extension (which you must enable) to perform high-speed geographic calculations. It finds the 50 shops closest to a given `lat` and `long`.

```
CREATE FUNCTION nearby_shops(lat float, long float)
RETURNS SETOF shops
LANGUAGE sql
AS $$
  SELECT *
  FROM shops
  ORDER BY (
    st_distance(
      st_point(longitude, latitude), -- Shop's location
      st_point(long, lat)         -- User's location
    )
  )
  LIMIT 50;
$$;
```

4. Part 3: Backend Server Logic (`index.js`)

This is the complete explanation of your server code, broken into logical pieces.

A. Initialization

These lines load all your tools and initialize the connections to Express, Supabase, Google AI, and Multer.

```
// --- 1. Load our tools ---

const express = require('express');

require('dotenv').config();

const { createClient } = require('@supabase/supabase-js');

const { GoogleGenerativeAI } = require('@google/generative-ai');

const multer = require('multer'); // For file uploads

// --- 2. Initialize the server and clients ---

const app = express();

const PORT = 3000;

console.log("Starting server setup...");

// Initialize Supabase

const supabaseUrl = process.env.SUPABASE_URL;

// ... (rest of initializations) ...

const supabase = createClient(supabaseUrl, supabaseKey);

// Initialize Google AI

const geminiApiKey = process.env.GEMINI_API_KEY;

// ... (rest of initializations) ...

const genAI = new GoogleGenerativeAI(geminiApiKey);

const model = genAI.getGenerativeModel({ model: "gemini-1.5-flash-latest" });

// Initialize Multer

const upload = multer({ storage: multer.memoryStorage() });

// Middleware to parse JSON bodies    app.use(express.json());
```

B. Reusable Helper Function: `findProducts`

This is the AI "brain" of your app. It's a single function that can be reused by both text search and image search.

```
async function findProducts(query, latitude, longitude) {
  // 1. Find nearby shops using the SQL function
  const { data: shops, error: shopsError } = await supabase.rpc('nearby_shops', {
    lat: latitude,
    long: longitude
  });
  // ... (error handling) ..
  // 2. Get all products from those nearby shops that match the text query
  const shopIds = shops.map(shop => shop.id);
  const { data: products, error: productsError } = await supabase
    .from('products')
    .select('*, shops (name, contact_number, email)') // Gets products AND their shop info
    .in('shop_id', shopIds)
    .ilike('name', `%${query}%`); // 'ilike' is a case-insensitive "contains" search
  // ... (error handling) ...
  // 3. Prepare a clean "context" list for the AI
  const contextForAI = products.map(p => ({
    product_name: p.name,
    price: p.price,
    rating: p.rating,
    shop_name: p.shops ? p.shops.name : 'Unknown Shop',
    contact_number: p.shops ? p.shops.contact_number : null,
```

```
    email: p.shops ? p.shops.email : null,
    shop_id: p.shop_id
  }));
  // 4. Create the prompt for the AI, inserting the data
  const prompt = `
    You are an expert shopping assistant... The user is searching for "${query}".
    Here is the list of available products:
    ${JSON.stringify(contextForAI)}
    ... (rest of the prompt instructions) ...
  `;
  // 5. Call the AI
  const result = await model.generateContent(prompt);
  const response = await result.response;
  const text = response.text();
  // 6. Clean and parse the AI's JSON text response
  const cleanedText = text.replace(/```json/g, "").replace(/```/g, "").trim();
  return JSON.parse(cleanedText);
}
```

C. Authentication Endpoints

These two endpoints allow users to sign up and log in. They are simple wrappers for Supabase's powerful built-in auth functions.

```
app.post('/auth/signup', async (req, res) => {
  try {
    const { email, password, full_name } = req.body;
    const { data, error } = await supabase.auth.signUp({
      email, password, options: { data: { full_name: full_name } }
    });
    if (error) throw error;
    res.status(201).json({ user: data.user, message: 'Signup successful!' });
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
});
app.post('/auth/login', async (req, res) => {
  try {
    const { email, password } = req.body;
    const { data, error } = await supabase.auth.signInWithPassword({ email, password });
    if (error) throw error;
    res.json({ session: data.session, user: data.user });
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
});
```

D. Security Middleware: `protectRoute`

This function is a "security guard" that you place in front of any endpoint that should only be accessed by a logged-in user. It reads the `Bearer Token` from the request header and validates it with Supabase.

```
const protectRoute = async (req, res, next) => {
  try {
    const authHeader = req.headers.authorization;
    if (!authHeader || !authHeader.startsWith('Bearer ')) {
      return res.status(401).json({ error: 'Auth header missing or malformed.' });
    }
    const token = authHeader.split(' ')[1]; // Get token from "Bearer <token>"
    const { data: { user }, error } = await supabase.auth.getUser(token);
    if (error || !user) {
      return res.status(401).json({ error: 'Invalid or expired token.' });
    }
    req.user = user; // IMPORTANT: Attach the user object to the request
    next(); // Proceed to the endpoint
  } catch (error) {
    res.status(401).json({ error: 'Authentication failed.' });
  }
};
```

E. Core Shop & Product Endpoints

These are the endpoints for creating shops and products. Notice they both include `protectRoute` in the function call, so only logged-in users can use them.

```
// Create a new shop
app.post('/shops', protectRoute, async (req, res) => {
  try {
    const { name, description, latitude, longitude, contact_number, email } = req.body;
    const owner_id = req.user.id; // Get the user ID from the security guard
    const { data, error } = await supabase
      .from('shops')
      .insert({ name, description, owner_id, latitude, longitude, contact_number, email })
      .select()
      .single();
    if (error) throw error;
    res.status(201).json(data);
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
});
// Create a new product
app.post('/products', protectRoute, async (req, res) => {
  try {
    const { name, description, price, shop_id, rating } = req.body;
    const { data, error } = await supabase
      .from('products')
```

```
      .insert({ name, description, price, shop_id, rating })
      .select()
      .single();
    if (error) throw error;
    res.status(201).json(data);
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
});
```

F. Feature 1: AI Text Search Endpoint

This endpoint is now extremely simple. It just validates the input (query, lat, long) and then calls our reusable `findProducts` helper to do all the work.

```
app.post('/ai/search', async (req, res) => {
  try {
    const { query, latitude, longitude } = req.body;
    if (!query || !latitude || !longitude) {
      return res.status(400).json({ error: 'Query, latitude, and longitude are required.' });
    }
    // Call the reusable helper function
    const results = await findProducts(query, latitude, longitude);
    res.json(results);
  } catch (error) {
    // ... (Error handling) ...
  }});
```

G. Feature 2: AI Image Search Endpoint

This endpoint uses `upload.single('foodImage')` to catch the file. It then performs two phases:

1. **Phase 1 (Identify):** Converts the image file to base64 text and sends it to Gemini to get the name of the food (e.g., "Burger").
2. **Phase 2 (Search):** It uses the returned `foodName` ("Burger") to call the *exact same* `findProducts` helper function, efficiently reusing your code.

```
app.post('/ai/search-by-image', upload.single('foodImage'), async (req, res) => {

  try {

    const { latitude, longitude } = req.body;

    const imageFile = req.file;

    // ... (Error handling for missing file/location) ...

    // 2. Prepare image for Gemini Vision

    const imagePart = {

      inlineData: {

        data: imageFile.buffer.toString("base64"),

        mimeType: imageFile.mimetype

      }

    };

    const prompt = "Identify the food in this image. Respond with ONLY the name...";

    // 3. Call Gemini to identify the food

    const result = await model.generateContent([prompt, imagePart]);

    const response = await result.response;

    const foodName = response.text().trim(); // foodName is now "Burger"
```

```
    // 4. Call the reusable findProducts function with the identified food name
    const searchResults = await findProducts(foodName, latitude, longitude);
    // 5. Return the results
    res.json(searchResults);
  } catch (error) {
    // ... (Error handling) ...
  }
});
```

NOT IMPLEMENTED YET , BUT STILL WORKING ON THIS.

H. Feature 3 & 4: Market Trend Endpoints

You have two endpoints for this feature: one to **add** price data and one to **analyze** it.

The `POST /market_price` endpoint lets you add new price points to your database.

```
app.post('/market_price', protectRoute, async (req, res) => {
    try {
        const { product_name, price, source } = req.body;
        if (!product_name || !price) {
            return res.status(400).json({ error: 'product_name and price are required.' });
        }
        const { data, error } = await supabase
            .from('market_price')
            .insert({ product_name, price, source: source || 'Internal' })
            .select()
            .single();
```

```
        if (error) throw error;
        res.status(201).json(data);
    } catch (error) {
        res.status(400).json({ error: error.message });
    }
});
```

The `GET /ai/market-trend` endpoint reads that data, performs its own simple analysis (latest vs. average), and then asks Gemini to provide a simple "BUY," "SELL," or "HOLD" prediction

```
app.get('/ai/market-trend/:productName', async (req, res) => {
  try {
    const { productName } = req.params;
    // 1. Get internal data from your 'market_price' table
    const { data: prices, error: dbError } = await supabase
      .from('market_price')
      // ... (query to get last 30 prices) ...
    if (dbError) throw dbError;
    // 2. Analyze internal trend (Latest vs. Average)
    let internalTrend = "Insufficient Data";
    if (prices && prices.length > 1) {
      // ... (logic to calculate if trend is Rising, Falling, or Stable) ...
    }
```

```
// 3. (Placeholder) Get external data
    const externalTrend = { trend: "Stable", source: "Online Market (Placeholder)" };
    // 4. Ask AI for a prediction based on the data
    const prompt = `
      You are a market analyst...
      Internal Data Trend: ${internalTrend}
      External Market Trend: ${externalTrend.trend}
      ... (rest of prompt asking for BUY/SELL/HOLD) ...
    `;
    const result = await model.generateContent(prompt);
    // ... (Clean and return AI's JSON response) ...
    res.json(JSON.parse(cleanedText));
  } catch (error) {
    // ... (Error handling) ...
  }
});
```

I. Server Start

This is the final command. It tells your server to "go live" and start listening for API calls on port 3000.

```
// --- 5. Start the server ---
app.listen(PORT, () => {
  console.log(`Server is successfully running on http://localhost:${PORT}`);
});
```