# Smart Shop Backend - Complete Technical Documentation

Version 1.0

Author: Koushik Chandaka & Gemini

Date: September 13, 2025

## **Table of Contents**

1. **Project Overview**
2. **Prerequisites**
3. **Part 1: The Foundation - Server & Database Setup**
   * 3.1 Initializing the Node.js Project
   * 3.2 Setting Up the Supabase Project
   * 3.3 The Initial Server Code (index.js)
   * 3.4 Testing the Foundation
4. **Part 2: User Authentication**
   * 4.1 Creating the profiles Table (SQL)
   * 4.2 The Automatic Profile Trigger (SQL Function)
   * 4.3 Security Policies for Profiles (SQL)
   * 4.4 Backend Code for Authentication (index.js)
   * 4.5 Troubleshooting: "Email not confirmed"
   * 4.6 Testing Authentication (Postman)
5. **Part 3: Shops and Products**
   * 5.1 Creating shops & products Tables (SQL)
   * 5.2 Security Policies for Shops & Products (SQL)
   * 5.3 Backend Code for Shops & Products (index.js)
   * 5.4 Testing Shops & Products (Postman)
6. **Part 4: AI-Powered Smart Search**
   * 6.1 Getting the Gemini API Key
   * 6.2 Preparing the Database for Location (PostGIS)
   * 6.3 The Final Backend Code (index.js)
   * 6.4 Testing the AI Smart Search (Postman)
7. **Final index.js Code**
8. **Complete Supabase SQL Setup Script**

### **1. Project Overview**

This document outlines the step-by-step process for building the backend server for the "Smart Shop" application. The backend is built using Node.js with the Express framework and a Supabase (PostgreSQL) database. It includes user authentication, shop and product management, and a powerful, location-aware smart search feature powered by Google's Gemini AI.

### **2. Prerequisites**

* **Node.js:** Installed on your machine.
* **Postman:** A tool for testing APIs.
* A **Supabase** account.
* A **Google AI Studio** account for the Gemini API key.

### **3. Part 1: The Foundation - Server & Database Setup**

#### **3.1 Initializing the Node.js Project**

1. Create a new folder: mkdir smart-shop-backend
2. Navigate into it: cd smart-shop-backend
3. Initialize the project: npm init -y
4. Install necessary packages: npm install express dotenv @supabase/supabase-js

#### **3.2 Setting Up the Supabase Project**

1. Create a new project on [Supabase.io](https://supabase.io).
2. Navigate to **Project Settings** > **API**.
3. Find your **Project URL** and anon public **API Key**.
4. In your project folder, create a file named .env.
5. Add your Supabase credentials to the .env file:  
   SUPABASE\_URL=YOUR\_SUPABASE\_URL  
   SUPABASE\_ANON\_KEY=YOUR\_SUPABASE\_ANON\_KEY

#### **3.3 The Initial Server Code (index.js)**

This code establishes a basic server and tests the database connection.

// index.js - Part 1  
const express = require('express');  
require('dotenv').config();  
const { createClient } = require('@supabase/supabase-js');  
  
const app = express();  
const PORT = 3000;  
  
const supabaseUrl = process.env.SUPABASE\_URL;  
const supabaseKey = process.env.SUPABASE\_ANON\_KEY;  
const supabase = createClient(supabaseUrl, supabaseKey);  
  
app.get('/', (req, res) => {  
 res.json({ message: "Success! Your Smart Shop backend is running." });  
});  
  
app.listen(PORT, () => {  
 console.log(`Server is successfully running on http://localhost:${PORT}`);  
});

#### **3.4 Testing the Foundation**

1. Run node index.js in your terminal.
2. Open a web browser and go to http://localhost:3000.
3. You should see: {"message":"Success! Your Smart Shop backend is running."}

### **4. Part 2: User Authentication**

#### **4.1 Creating the profiles Table (SQL)**

In the Supabase SQL Editor, run this to create the table for user profile data.

create table profiles (  
 id uuid references auth.users not null primary key,  
 full\_name text,  
 created\_at timestamp with time zone default timezone('utc'::text, now()) not null  
);

#### **4.2 The Automatic Profile Trigger (SQL Function)**

This is the definitive solution to automatically create a profile when a new user signs up. Run in the SQL Editor.

-- Create a function that automatically creates a profile for a new user.  
create function public.handle\_new\_user()  
returns trigger  
language plpgsql  
security definer set search\_path = public  
as $$  
begin  
 insert into public.profiles (id, full\_name)  
 values (new.id, new.raw\_user\_meta\_data->>'full\_name');  
 return new;  
end;  
$$;  
  
-- Create a trigger that runs the function after a new user signs up.  
create trigger on\_auth\_user\_created  
 after insert on auth.users  
 for each row execute procedure public.handle\_new\_user();

#### **4.3 Security Policies for Profiles (SQL)**

These rules secure the profiles table. Run in the SQL Editor.

-- Enable Row Level Security  
alter table public.profiles enable row level security;  
  
-- Allow users to view their OWN profile.  
create policy "Users can view their own profile."  
on public.profiles for select  
using ( auth.uid() = id );  
  
-- Allow the trigger function to INSERT new profiles.  
create policy "Allow new user profile creation"  
on public.profiles for insert  
with check (true);

#### **4.4 Backend Code for Authentication (index.js)**

The index.js file is updated to handle /signup and /login routes. This code is included in the final version in Section 7.

#### **4.5 Troubleshooting: "Email not confirmed"**

By default, Supabase requires email verification. To speed up development:

1. Go to **Authentication** > **Providers** > **Email**.
2. Toggle **"Confirm email"** to OFF.
3. Click **Save**.

#### **4.6 Testing Authentication (Postman)**

**Signup Request:**

* **Method:** POST
* **URL:** http://localhost:3000/auth/signup
* **Body** (raw, JSON):  
  {  
   "email": "your-email@example.com",  
   "password": "your-strong-password",  
   "full\_name": "Your Name"  
  }
* **Expected Response:** 201 Created

**Login Request:**

* **Method:** POST
* **URL:** http://localhost:3000/auth/login
* **Body** (raw, JSON):  
  {  
   "email": "your-email@example.com",  
   "password": "your-strong-password"  
  }
* **Expected Response:** 200 OK with a session object containing an access\_token.

### **5. Part 3: Shops and Products**

#### **5.1 Creating shops & products Tables (SQL)**

Run in the SQL Editor.

-- Table for Shops  
create table shops (  
 id bigint generated by default as identity primary key,  
 created\_at timestamp with time zone not null default now(),  
 name text not null,  
 description text,  
 latitude numeric,  
 longitude numeric,  
 owner\_id uuid not null references public.profiles(id)  
);  
  
-- Table for Products  
create table products (  
 id bigint generated by default as identity primary key,  
 created\_at timestamp with time zone not null default now(),  
 name text not null,  
 description text,  
 price numeric not null,  
 shop\_id bigint not null references public.shops(id)  
);

#### **5.2 Security Policies for Shops & Products (SQL)**

Run in the SQL Editor.

alter table public.shops enable row level security;  
alter table public.products enable row level security;  
  
create policy "Authenticated users can create shops." on public.shops for insert to authenticated with check (true);  
create policy "Shop owners can view their own shop." on public.shops for select using ( auth.uid() = owner\_id );  
create policy "Shop owners can update their own shop." on public.shops for update using ( auth.uid() = owner\_id );  
create policy "Anyone can view shops." on public.shops for select using (true);  
  
create policy "Shop owners can add products." on public.products for insert with check ( exists (select 1 from shops where shops.id = products.shop\_id and shops.owner\_id = auth.uid()));  
create policy "Shop owners can manage their products." on public.products for select, update, delete using ( exists (select 1 from shops where shops.id = products.shop\_id and shops.owner\_id = auth.uid()));  
create policy "Anyone can view products." on public.products for select using (true);

#### **5.3 Backend Code for Shops & Products (index.js)**

The index.js file is updated to handle /shops and /products routes with a protectRoute middleware. This code is included in the final version in Section 7.

#### **5.4 Testing Shops & Products (Postman)**

1. First, /login to get a valid access\_token.
2. Copy the token.

**Create Shop Request:**

* **Method:** POST
* **URL:** http://localhost:3000/shops
* **Authorization:** Type Bearer Token, paste your token.
* **Body** (raw, JSON):  
  {  
   "name": "My Awesome General Store",  
   "description": "Selling the best goods online."  
  }
* **Expected Response:** 201 Created. Note the id of the new shop.

**Create Product Request:**

* **Method:** POST
* **URL:** http://localhost:3000/products
* **Authorization:** Type Bearer Token, paste your token.
* **Body** (raw, JSON), using the id from the created shop:  
  {  
   "name": "Super Widget",  
   "price": 99.99,  
   "shop\_id": 1   
  }
* **Expected Response:** 201 Created.

### **6. Part 4: AI-Powered Smart Search**

#### **6.1 Getting the Gemini API Key**

1. Go to [Google AI Studio](https://aistudio.google.com/app/apikey).
2. Click **"Create API key in new project"**.
3. Copy the key.
4. Add it to your .env file:  
   GEMINI\_API\_KEY=AIzaSy...your...key...here

#### **6.2 Preparing the Database for Location (PostGIS)**

1. In the Supabase dashboard, go to **Database** > **Extensions**.
2. Search for postgis and enable it.
3. In the **SQL Editor**, run the function to find nearby shops:  
   create function nearby\_shops(lat float, long float)  
   returns setof shops  
   language sql  
   as $$  
    select \*  
    from shops  
    order by (  
    st\_distance(  
    st\_point(longitude, latitude),  
    st\_point(long, lat)  
    )  
    )  
    limit 50;  
   $$;

#### **6.3 The Final Backend Code (index.js)**

The final code integrates the Google AI client and the /ai/search endpoint. See the complete code in Section 7.

#### **6.4 Testing the AI Smart Search (Postman)**

1. First, update a shop in your database with a real latitude and longitude via the Supabase Table Editor.
2. Restart your server.

**AI Search Request:**

* **Method:** POST
* **URL:** http://localhost:3000/ai/search
* **Body** (raw, JSON):  
  {  
   "query": "Burger",  
   "latitude": 19.0760,  
   "longitude": 72.8777  
  }
* **Expected Response:** 200 OK with a JSON object from the AI containing a summary and recommendations.

### **7. Final index.js Code**

This is the complete, final code for your server.

// --- 1. Load our tools ---  
const express = require('express');  
require('dotenv').config();  
const { createClient } = require('@supabase/supabase-js');  
const { GoogleGenerativeAI } = require('@google/generative-ai');  
  
// --- 2. Initialize the server and clients ---  
const app = express();  
const PORT = 3000;  
  
console.log("Starting server setup...");  
  
const supabaseUrl = process.env.SUPABASE\_URL;  
const supabaseKey = process.env.SUPABASE\_ANON\_KEY;  
const supabase = createClient(supabaseUrl, supabaseKey);  
console.log("Supabase client initialized.");  
  
const geminiApiKey = process.env.GEMINI\_API\_KEY;  
const genAI = new GoogleGenerativeAI(geminiApiKey);  
const model = genAI.getGenerativeModel({ model: "gemini-1.5-flash-latest" });  
console.log("Google AI client initialized.");  
  
app.use(express.json());  
  
// --- 3. Define our server's behavior (API Endpoints) ---  
app.get('/', (req, res) => {  
 res.json({ message: "Success! Your Smart Shop backend is running." });  
});  
  
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! Trigger will create profile.' });  
 } catch (error) {  
 console.error("Signup Error:", error.message);  
 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) {  
 console.error("Login Error:", error.message);  
 res.status(400).json({ error: error.message });  
 }  
});  
  
const protectRoute = async (req, res, next) => {  
 try {  
 const authHeader = req.headers.authorization;  
 if (!authHeader || !authHeader.startsWith('Bearer ')) {  
 return res.status(401).json({ error: 'Authorization header is missing or malformed.' });  
 }  
 const token = authHeader.split(' ')[1];  
 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;  
 next();  
 } catch (error) {  
 res.status(401).json({ error: 'Authentication failed.' });  
 }  
};  
  
app.post('/shops', protectRoute, async (req, res) => {  
 try {  
 const { name, description, latitude, longitude } = req.body;  
 const owner\_id = req.user.id;  
 const { data, error } = await supabase  
 .from('shops')  
 .insert({ name, description, owner\_id, latitude, longitude })  
 .select()  
 .single();  
 if (error) throw error;  
 res.status(201).json(data);  
 } catch (error) {  
 console.error("Create Shop Error:", error.message);  
 res.status(400).json({ error: error.message });  
 }  
});  
  
app.get('/shops', async (req, res) => {  
 try {  
 const { data, error } = await supabase.from('shops').select('\*');  
 if (error) throw error;  
 res.json(data);  
 } catch (error) {  
 res.status(500).json({ error: error.message });  
 }  
});  
  
app.post('/products', protectRoute, async (req, res) => {  
 try {  
 const { name, description, price, shop\_id } = req.body;  
 const { data, error } = await supabase  
 .from('products')  
 .insert({ name, description, price, shop\_id })  
 .select()  
 .single();  
 if (error) throw error;  
 res.status(201).json(data);  
 } catch (error) {  
 console.error("Add Product Error:", error.message);  
 res.status(400).json({ error: error.message });  
 }  
});  
  
app.get('/products', async (req, res) => {  
 try {  
 let query = supabase.from('products').select('\*');  
 if (req.query.shop\_id) {  
 query = query.eq('shop\_id', req.query.shop\_id);  
 }  
 const { data, error } = await query;  
 if (error) throw error;  
 res.json(data);  
 } catch (error) {  
 res.status(500).json({ error: error.message });  
 }  
});  
  
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.' });  
 }  
  
 const { data: shops, error: shopsError } = await supabase.rpc('nearby\_shops', { lat: latitude, long: longitude });  
 if (shopsError) throw shopsError;  
  
 if (!shops || shops.length === 0) {  
 return res.json({ summary: "No shops found near your location.", recommendations: [] });  
 }  
  
 const shopIds = shops.map(shop => shop.id);  
 const { data: products, error: productsError } = await supabase  
 .from('products')  
 .select('\*, shops (name)')  
 .in('shop\_id', shopIds)  
 .ilike('name', `%${query}%`);  
 if (productsError) throw productsError;  
  
 if (!products || products.length === 0) {  
 return res.json({ summary: `No products matching "${query}" found in nearby shops.`, recommendations: [] });  
 }  
   
 const contextForAI = products.map(p => ({  
 product\_name: p.name,  
 price: p.price,  
 shop\_name: p.shops ? p.shops.name : 'Unknown Shop',  
 shop\_id: p.shop\_id  
 }));  
  
 const prompt = `  
 You are an expert shopping assistant for an app called 'Smart Shop'. Your task is to analyze a list of products found near a user and recommend the best options. The user is searching for "${query}".  
 Here is the list of available products from nearby shops:  
 ${JSON.stringify(contextForAI)}  
 Analyze the list and provide a helpful summary. Then, identify the top 3 best options. Rank them by a combination of best price and quality. For each of the top 3 recommendations, explain briefly why it is a good choice.  
 Respond ONLY with a valid JSON object in the following format. Do not include any other text or markdown formatting like \`\`\`json.  
 {  
 "summary": "Your overall analysis here. Be friendly and helpful.",  
 "recommendations": [  
 {  
 "rank": 1,  
 "product\_name": "Product Name",  
 "price": 199.50,  
 "shop\_name": "Shop Name",  
 "reason": "Brief reason for recommendation."  
 }  
 ]  
 }  
 `;  
  
 const result = await model.generateContent(prompt);  
 const response = await result.response;  
 const text = response.text();  
   
 const cleanedText = text.replace(/```json/g, "").replace(/```/g, "").trim();  
 res.json(JSON.parse(cleanedText));  
  
 } catch (error) {  
 console.error("\n--- !!! FULL AI SEARCH ERROR !!! ---");  
 console.error(error);  
 console.error("--- !!! END OF ERROR !!! ---\n");  
 if (error instanceof SyntaxError && error.message.includes('JSON')) {  
 return res.status(500).json({ error: 'The AI returned a response that was not valid JSON.' });  
 }  
 res.status(500).json({ error: 'An internal error occurred. Please check the server logs for the full error details.' });  
 }  
});  
  
// --- 4. Start the server ---  
app.listen(PORT, () => {  
 console.log(`Server is successfully running on http://localhost:${PORT}`);  
});

### **8. Complete Supabase SQL Setup Script**

You can run this entire script in the Supabase SQL Editor on a fresh project to set up the entire database structure and security in one go.

-- 1. Create Tables  
create table profiles (  
 id uuid references auth.users not null primary key,  
 full\_name text,  
 created\_at timestamp with time zone default timezone('utc'::text, now()) not null  
);  
  
create table shops (  
 id bigint generated by default as identity primary key,  
 created\_at timestamp with time zone not null default now(),  
 name text not null,  
 description text,  
 latitude numeric,  
 longitude numeric,  
 owner\_id uuid not null references public.profiles(id)  
);  
  
create table products (  
 id bigint generated by default as identity primary key,  
 created\_at timestamp with time zone not null default now(),  
 name text not null,  
 description text,  
 price numeric not null,  
 shop\_id bigint not null references public.shops(id)  
);  
  
-- 2. Create Automatic Profile Function & Trigger  
create function public.handle\_new\_user()  
returns trigger  
language plpgsql  
security definer set search\_path = public  
as $$  
begin  
 insert into public.profiles (id, full\_name)  
 values (new.id, new.raw\_user\_meta\_data->>'full\_name');  
 return new;  
end;  
$$;  
  
create trigger on\_auth\_user\_created  
 after insert on auth.users  
 for each row execute procedure public.handle\_new\_user();  
  
-- 3. Setup Security Policies  
alter table public.profiles enable row level security;  
create policy "Users can view their own profile." on public.profiles for select using ( auth.uid() = id );  
create policy "Allow new user profile creation" on public.profiles for insert with check (true);  
  
alter table public.shops enable row level security;  
create policy "Authenticated users can create shops." on public.shops for insert to authenticated with check (true);  
create policy "Shop owners can view their own shop." on public.shops for select using ( auth.uid() = owner\_id );  
create policy "Shop owners can update their own shop." on public.shops for update using ( auth.uid() = owner\_id );  
create policy "Anyone can view shops." on public.shops for select using (true);  
  
alter table public.products enable row level security;  
create policy "Shop owners can add products." on public.products for insert with check ( exists (select 1 from shops where shops.id = products.shop\_id and shops.owner\_id = auth.uid()));  
create policy "Shop owners can manage their products." on public.products for select, update, delete using ( exists (select 1 from shops where shops.id = products.shop\_id and shops.owner\_id = auth.uid()));  
create policy "Anyone can view products." on public.products for select using (true);  
  
-- 4. Create Location Search Function (Requires PostGIS extension to be enabled)  
create function nearby\_shops(lat float, long float)  
returns setof shops  
language sql  
as $$  
 select \*  
 from shops  
 order by (  
 st\_distance(  
 st\_point(longitude, latitude),  
 st\_point(long, lat)  
 )  
 )  
 limit 50;  
$$;

## **End of Document**