######

!pip install --quiet streamlit langchain langchain-google-genai google-generativeai python-dotenv pyngrok googlemaps geopy pandas

######  
# We’ll safely collect keys without printing them

from getpass import getpass

import textwrap, os

# Ask once; you can re-run this cell any time to change keys

GEMINI\_KEY = getpass("Enter your GOOGLE\_API\_KEY (Gemini): ")

MAPS\_KEY  = getpass("Enter your MAPS\_API\_KEY (Google Maps) [press Enter to skip]: ")

# Create a .env file so the rest of the app can load keys automatically

env\_text = textwrap.dedent(f"""

GOOGLE\_API\_KEY={GEMINI\_KEY}

MAPS\_API\_KEY={MAPS\_KEY}

""").strip()

# Write the .env file in the current working directory

with open(".env", "w") as f:

    f.write(env\_text)

# Confirm the file exists (we never print your actual keys)

print("✅ .env written (keys NOT shown).")  
  
######

%%writefile agent.py

# ------------------------------ agent.py ------------------------------

# This file implements ALL 6 requested features for FindMyStore:

# 1) Nearby Store Finder + Maps/filters

# 2) Product Availability + Price Comparison

# 3) Smart Shopping List + Budget Mode

# 4) Real-Time Stock & Deal Alerts (demo)

# 5) Voice + AI Chatbot (Gemini) that can call tools

# 6) Community Reviews & Verified Stores (from rating or simulated)

import os, json, random

from typing import List, Dict, Any, Tuple

# Load API keys from .env

from dotenv import load\_dotenv

load\_dotenv()

GOOGLE\_API\_KEY = os.getenv("GOOGLE\_API\_KEY", "")   # Gemini key

MAPS\_API\_KEY   = os.getenv("MAPS\_API\_KEY", "")     # Google Maps key (optional for demo)

# LangChain + Gemini LLM

from langchain\_google\_genai import ChatGoogleGenerativeAI

from langchain.agents import initialize\_agent, Tool, AgentType

# Utilities for geo + data

import pandas as pd

from geopy.distance import geodesic

# Google Maps client (if key provided)

try:

    import googlemaps

    gmaps = googlemaps.Client(key=MAPS\_API\_KEY) if MAPS\_API\_KEY else None

except Exception:

    gmaps = None

# --------- Category mapping (Dialogflow-friendly naming -> Places types) ----------

CATEGORY\_TO\_TYPE = {

    "grocery": "grocery\_or\_supermarket",

    "pharmacy": "pharmacy",

    "electronics": "electronics\_store",

    "clothing": "clothing\_store",

    "bakery": "bakery",

    "restaurant": "restaurant",

}

# ---------------- Fallback (simulated) store data if Maps API not used ----------------

stores\_fallback = [

    {"id": 1, "name": "SmartMart Jubilee Hills", "city": "Hyderabad", "category": "grocery",

     "hours": "9am-9pm", "lat": 17.433, "lng": 78.403, "rating": 4.5, "verified": True},

    {"id": 2, "name": "MediCare Pharmacy Banjara", "city": "Hyderabad", "category": "pharmacy",

     "hours": "24/7", "lat": 17.412, "lng": 78.448, "rating": 4.3, "verified": True},

    {"id": 3, "name": "ElectroHub Secunderabad", "city": "Hyderabad", "category": "electronics",

     "hours": "10am-8pm", "lat": 17.444, "lng": 78.501, "rating": 4.1, "verified": False},

    {"id": 4, "name": "StyleStreet Hitech City", "city": "Hyderabad", "category": "clothing",

     "hours": "11am-9pm", "lat": 17.452, "lng": 78.381, "rating": 4.6, "verified": True},

]

# -------------- Inventory with price (qty + ₹ price). Keys are store IDs ---------------

inventory: Dict[int, Dict[str, Dict[str, float]]] = {

    1: {"XYZ Shampoo": {"qty": 12, "price": 150}, "Milk Lotion": {"qty": 0, "price": 199}, "Rice 10kg": {"qty": 8, "price": 489}},

    2: {"XYZ Shampoo": {"qty":  4, "price": 155}, "Milk Lotion": {"qty": 7, "price": 189}},

    3: {"Laptop Bag":  {"qty":  6, "price": 899}, "USB Cable":   {"qty": 15, "price": 149}},

    4: {"T-Shirt":     {"qty": 10, "price": 399}},

}

# Dynamic inventory for live-fetched stores (created on demand)

dynamic\_inventory: Dict[int, Dict[str, Dict[str, float]]] = {}

# Cache of the most recently fetched stores (for compare & shopping list)

\_last\_store\_cache: List[Dict[str, Any]] = []

# --------------------------- Helpers: geo + links --------------------------------

def geocode\_city(city: str) -> Tuple[float, float]:

    """Geocode city name -> (lat, lng). Falls back to Hyderabad center if Maps disabled."""

    if gmaps:

        ge = gmaps.geocode(city)

        if ge and ge[0]["geometry"]["location"]:

            loc = ge[0]["geometry"]["location"]

            return loc["lat"], loc["lng"]

    return (17.3850, 78.4867)  # Hyderabad fallback

def maps\_place\_link(lat: float, lng: float, place\_id: str = None) -> str:

    """Return a Google Maps link to a place or lat/lng."""

    if place\_id:

        return f"https://www.google.com/maps/search/?api=1&query=Google&query\_place\_id={place\_id}"

    return f"https://www.google.com/maps?q={lat},{lng}"

def maps\_directions\_link(dest\_lat: float, dest\_lng: float, place\_id: str = None, origin: str = None) -> str:

    """Return a Google Maps directions URL."""

    base = "https://www.google.com/maps/dir/?api=1"

    parts = [base, f"destination={dest\_lat},{dest\_lng}"]

    if place\_id:

        parts.append(f"destination\_place\_id={place\_id}")

    if origin:

        parts.append(f"origin={origin}")

    parts.append("travelmode=driving")

    return "&".join(parts)

# --------------------- Fetch stores (Maps API or fallback) -----------------------

def fetch\_stores(city: str, category: str = None, radius\_km: float = 6.0, open\_now: bool = False) -> List[Dict[str, Any]]:

    """Fetch stores by city/category + open status. Uses Google Maps if available, else simulated."""

    global \_last\_store\_cache

    results: List[Dict[str, Any]] = []

    if gmaps:

        lat, lng = geocode\_city(city)

        ptype = CATEGORY\_TO\_TYPE.get(category or "", "store")  # "store" is broad fallback

        places = gmaps.places\_nearby(

            location=(lat, lng),

            radius=int(radius\_km \* 1000),

            type=ptype if ptype != "store" else None,  # None for broad search

            open\_now=open\_now if ptype != "store" else None,

        )

        pid = 1

        for p in places.get("results", []):

            loc = p.get("geometry", {}).get("location", {})

            rating = float(p.get("rating", 4.2))

            urt = int(p.get("user\_ratings\_total", 20))

            results.append({

                "id": pid,                                  # Local sequential ID for our UI

                "place\_id": p.get("place\_id"),

                "name": p.get("name"),

                "city": city,

                "category": category or "general",

                "hours": "—",

                "lat": loc.get("lat"),

                "lng": loc.get("lng"),

                "address": p.get("vicinity", ""),

                "rating": rating,

                "verified": True if (rating >= 4.3 and urt >= 30) else False,  # Simple rule

            })

            pid += 1

    else:

        # Simulated fallback results

        results = [s for s in stores\_fallback if s["city"].lower() == city.lower()]

        if category:

            results = [s for s in results if s["category"] == category]

    \_last\_store\_cache = results[:]  # cache for later use

    return results

# ------------------------- Inventory helpers ------------------------------------

def ensure\_dynamic\_inventory(store\_id: int, product: str):

    """Give a live-fetched store a made-up stock+price so the demo feels real."""

    di = dynamic\_inventory.setdefault(store\_id, {})

    if product not in di:

        di[product] = {"qty": random.randint(0, 12),

                       "price": random.choice([79, 99, 129, 149, 179, 199, 249, 299, 349, 399, 899, 999])}

def get\_inventory(store\_id: int) -> Dict[str, Dict[str, float]]:

    """Return inventory dict (fallback + dynamic)."""

    if store\_id in inventory:

        return inventory[store\_id]

    return dynamic\_inventory.setdefault(store\_id, {})

def check\_stock(store\_id: int, product: str) -> Dict[str, Any]:

    """Return {'qty', 'price'} for a product in a store."""

    inv = get\_inventory(store\_id)

    if product not in inv:

        ensure\_dynamic\_inventory(store\_id, product)

        inv = get\_inventory(store\_id)

    data = inv.get(product, {"qty": 0, "price": None})

    return {"store\_id": store\_id, "product": product, "qty": int(data["qty"]), "price": data["price"]}

def find\_cheapest(product: str, max\_price: float = None) -> Dict[str, Any]:

    """Search cached stores for cheapest available product (with optional max price)."""

    search\_space = \_last\_store\_cache[:] or stores\_fallback[:]

    best = None

    for s in search\_space:

        info = check\_stock(s["id"], product)

        if info["qty"] > 0 and info["price"] is not None:

            if max\_price is not None and info["price"] > max\_price:

                continue

            if best is None or info["price"] < best["price"]:

                best = {\*\*info, "store": s}

    return best or {"message": f"No available '{product}' found within criteria."}

def shopping\_list\_optimize(products: List[str]) -> Dict[str, Any]:

    """Greedy set-cover style plan to buy a list of items at minimum total cost across stores."""

    needed = set([p.strip() for p in products if p.strip()])

    if not needed:

        return {"message": "No items in shopping list."}

    search\_space = \_last\_store\_cache[:] or stores\_fallback[:]

    offerings = {}

    for s in search\_space:

        sid = s["id"]

        o = {}

        for item in needed:

            info = check\_stock(sid, item)

            if info["qty"] > 0 and info["price"] is not None:

                o[item] = info["price"]

        if o:

            offerings[sid] = o

    chosen = []

    remaining = set(needed)

    total\_cost = 0.0

    while remaining:

        best\_sid, best\_cover, best\_score = None, set(), float("inf")

        for sid, offer in offerings.items():

            cover = remaining.intersection(offer.keys())

            if cover:

                cost = sum(offer[i] for i in cover)

                score = cost / len(cover)  # lower is better

                if score < best\_score:

                    best\_sid, best\_cover, best\_score = sid, cover, score

        if not best\_sid:

            break  # cannot cover all items

        chosen.append({

            "store\_id": best\_sid,

            "items": {i: offerings[best\_sid][i] for i in best\_cover},

            "subtotal": sum(offerings[best\_sid][i] for i in best\_cover),

        })

        total\_cost += chosen[-1]["subtotal"]

        remaining -= best\_cover

        offerings.pop(best\_sid, None)

    return {

        "covered\_all": len(remaining) == 0,

        "plan": chosen,

        "not\_found": list(remaining),

        "total\_cost": round(total\_cost, 2),

    }

# ---------------------- Alerts (demo: in-memory) ----------------------

\_subscriptions: List[Dict[str, Any]] = []  # e.g., {"product": "XYZ Shampoo", "city": "Hyderabad"}

def subscribe\_alert(product: str, city: str):

    """Remember what the user wants to be alerted for (demo only)."""

    \_subscriptions.append({"product": product.strip(), "city": city.strip()})

    return {"message": f"Subscribed to '{product}' alerts in {city} (demo)."}

def simulate\_restock(product: str, store\_id: int, qty: int = 10):

    """Pretend the store restocked; update inventory and 'notify' subscribers (demo)."""

    inv = get\_inventory(store\_id)

    if product not in inv:

        inv[product] = {"qty": qty, "price": random.choice([99, 129, 149, 199, 249, 299])}

    else:

        inv[product]["qty"] = max(0, inv[product]["qty"]) + qty

    hits = [s for s in \_subscriptions if s["product"].lower() == product.lower()]

    return {"message": f"Restocked '{product}' at store {store\_id}. Alerts notified: {len(hits)} (demo)."}

# -------------------- Public helpers for Streamlit UI -----------------

def ui\_search\_stores(city: str, category: str = None, radius\_km: float = 6.0, open\_now: bool = False):

    """UI wrapper for fetching stores."""

    return fetch\_stores(city, category, radius\_km, open\_now)

def ui\_compare\_prices(product: str, city: str = None, category: str = None, radius\_km: float = 6.0):

    """UI wrapper that returns a DataFrame of availability+price across stores."""

    if city:

        fetch\_stores(city, category, radius\_km)  # refresh cache

    rows = []

    for s in \_last\_store\_cache or stores\_fallback:

        info = check\_stock(s["id"], product)

        rows.append({

            "store\_id": s["id"],

            "store": s.get("name"),

            "city": s.get("city"),

            "category": s.get("category"),

            "verified": s.get("verified", False),

            "rating": s.get("rating"),

            "qty": info["qty"],

            "price": info["price"],

            "map": maps\_place\_link(s.get("lat"), s.get("lng"), s.get("place\_id")),

        })

    df = pd.DataFrame(rows)

    if not df.empty:

        df = df.sort\_values(by=["qty","price"], ascending=[False, True])

    return df

def ui\_shopping\_list(products\_csv: str, city: str = None, category: str = None, radius\_km: float = 6.0):

    """UI wrapper for shopping list optimization."""

    if city:

        fetch\_stores(city, category, radius\_km)

    items = [x.strip() for x in products\_csv.split(",") if x.strip()]

    return shopping\_list\_optimize(items)

def ui\_get\_directions(store\_id: int, origin: str = None) -> str:

    """Return a directions link for the chosen store."""

    s = next((x for x in (\_last\_store\_cache or stores\_fallback) if x["id"] == store\_id), None)

    if not s:

        return "Store not found."

    return maps\_directions\_link(s["lat"], s["lng"], s.get("place\_id"), origin)

# ------------------------ LangChain Agent (Gemini) --------------------

llm = ChatGoogleGenerativeAI(

    model="models/gemini-1.5-pro-latest",  # Gemini 1.5 Pro

    temperature=0.3,

    google\_api\_key=GOOGLE\_API\_KEY,

)

# Tool: find stores (input: "city|category|radius\_km|open\_now")

def \_tool\_find\_stores(q: str) -> str:

    parts = [p.strip() or None for p in q.split("|")]

    city = parts[0] if len(parts) > 0 else "Hyderabad"

    category = parts[1] if len(parts) > 1 else None

    radius = float(parts[2]) if len(parts) > 2 and parts[2] else 6.0

    open\_now = (parts[3].lower() == "true") if len(parts) > 3 and parts[3] else False

    out = fetch\_stores(city, category, radius, open\_now)

    return json.dumps(out, ensure\_ascii=False)

# Tool: check inventory (input: "store\_id|product")

def \_tool\_check\_inventory(q: str) -> str:

    sid, product = q.split("|", 1)

    out = check\_stock(int(sid), product.strip())

    return json.dumps(out, ensure\_ascii=False)

# Tool: find cheapest (input: "product|max\_price(optional)")

def \_tool\_find\_cheapest(q: str) -> str:

    parts = [p.strip() for p in q.split("|")]

    product = parts[0]

    max\_price = float(parts[1]) if len(parts) > 1 and parts[1] else None

    out = find\_cheapest(product, max\_price)

    return json.dumps(out, ensure\_ascii=False)

# Tool: optimize shopping list (input: "item1,item2,...")

def \_tool\_shopping\_list(q: str) -> str:

    items = [x.strip() for x in q.split(",") if x.strip()]

    return json.dumps(shopping\_list\_optimize(items), ensure\_ascii=False)

# Tool: directions link (input: "store\_id|origin(optional)")

def \_tool\_get\_directions(q: str) -> str:

    parts = [p.strip() for p in q.split("|")]

    sid = int(parts[0])

    origin = parts[1] if len(parts) > 1 and parts[1] else None

    return ui\_get\_directions(sid, origin)

# Tool: subscribe alerts (input: "product|city")

def \_tool\_subscribe\_alert(q: str) -> str:

    product, city = q.split("|", 1)

    return json.dumps(subscribe\_alert(product, city), ensure\_ascii=False)

# Register tools with the agent

tools = [

    Tool(name="FindStoreTool",       func=\_tool\_find\_stores,

         description="Find stores by city/category. Input: 'city|category|radius\_km|open\_now' (open\_now true/false)."),

    Tool(name="CheckInventoryTool",  func=\_tool\_check\_inventory,

         description="Check product stock/price in a store. Input: 'store\_id|product'."),

    Tool(name="FindCheapestTool",    func=\_tool\_find\_cheapest,

         description="Find the cheapest store for a product. Input: 'product|max\_price(optional)'."),

    Tool(name="ShoppingListTool",    func=\_tool\_shopping\_list,

         description="Optimize shopping list across stores. Input: 'item1,item2,...'."),

    Tool(name="GetDirectionsTool",   func=\_tool\_get\_directions,

         description="Get Google Maps directions link. Input: 'store\_id|origin(optional)'."),

    Tool(name="SubscribeAlertTool",  func=\_tool\_subscribe\_alert,

         description="Subscribe to restock alerts (demo). Input: 'product|city'."),

]

# Use Conversational agent so it remembers context in a session

agent = initialize\_agent(

    tools=tools,

    llm=llm,

    agent=AgentType.CONVERSATIONAL\_REACT\_DESCRIPTION,

    verbose=False,

)

# Keep chat history inside Python memory so context works

chat\_history: list[tuple[str, str]] = []

def run\_agent(query: str) -> str:

    global chat\_history

    try:

        response = agent.invoke({

            "input": query,

            "chat\_history": chat\_history

        })

        if isinstance(response, dict):

            answer = response.get("output", response.get("output\_text", str(response)))

        else:

            answer = str(response)

        # store turn in memory

        chat\_history.append(("user", query))

        chat\_history.append(("assistant", answer))

        return answer

    except Exception as e:

        return f"Agent error: {e}"

# ------------------------------ end agent.py ------------------------------

######  
%%writefile app.py

# ------------------------------ app.py ------------------------------

# Streamlit front-end with 5 tabs:

# 1) Nearby Stores (map + filters)          2) Product Compare

# 3) Shopping List Optimizer                4) Alerts (demo)

# 5) Chatbot (Gemini + tools)

import streamlit as st

import pandas as pd

# Import UI functions and chatbot from agent.py

from agent import (

    ui\_search\_stores, ui\_compare\_prices, ui\_shopping\_list,

    ui\_get\_directions, run\_agent, subscribe\_alert, simulate\_restock

)

# Basic page settings

st.set\_page\_config(page\_title="FindMyStore", layout="wide")

st.title("🏬 FindMyStore — Demo")

# Sidebar filters — applied to multiple tabs

with st.sidebar:

    st.header("Search Filters")

    city = st.text\_input("City", value="Hyderabad")

    category = st.selectbox("Category", ["", "grocery", "pharmacy", "electronics", "clothing", "bakery", "restaurant"])

    radius = st.slider("Radius (km)", 1, 20, 6)

    open\_now = st.checkbox("Open now", value=False)

    st.caption("Tip: Leave category blank for a broad search.")

# Tabs for features

tabs = st.tabs(["🗺️ Nearby Stores", "🔍 Product Compare", "🛒 Shopping List", "🔔 Alerts (Demo)", "🤖 Chatbot"])

# -------------------- Tab 1: Nearby Stores --------------------

with tabs[0]:

    st.subheader("Nearby Store Finder + Maps Integration")

    if st.button("Find Stores", type="primary"):

        results = ui\_search\_stores(city, category or None, radius, open\_now)

        if not results:

            st.warning("No stores found. Try a different category or radius.")

        else:

            st.success(f"Found {len(results)} store(s).")

            # Map: expects DataFrame with lat/lon columns

            df\_map = pd.DataFrame([{"lat": r["lat"], "lon": r["lng"]} for r in results if r.get("lat") and r.get("lng")])

            if not df\_map.empty:

                st.map(df\_map, size=100)

            # Render cards with main details

            for s in results:

                with st.container():

                    left, right = st.columns([0.7, 0.3])

                    with left:

                        badge = "✅ Verified" if s.get("verified") else "⚪ Unverified"

                        stars = "⭐" \* int(round(s.get("rating", 4.0)))

                        st.markdown(f"\*\*{s['name']}\*\*  \n{badge} · {stars}")

                        st.write(f"Category: {s.get('category','—')}  |  City: {s.get('city','—')}")

                        if s.get("address"):

                            st.write(f"Address: {s['address']}")

                        if s.get("hours"):

                            st.write(f"Hours: {s['hours']}")

                    with right:

                        link = f"https://www.google.com/maps?q={s['lat']},{s['lng']}"

                        st.link\_button("📍 View on Map", link)

# -------------------- Tab 2: Product Compare --------------------

with tabs[1]:

    st.subheader("Product Availability + Price Comparison")

    col1, col2 = st.columns([0.6,0.4])

    with col1:

        product = st.text\_input("Product name", value="XYZ Shampoo")

        if st.button("Compare Prices"):

            df = ui\_compare\_prices(product, city, category or None, radius)

            if df.empty:

                st.warning("No data.")

            else:

                # Highlight the cheapest available option

                cheapest = df[df["qty"] > 0].sort\_values(by="price", ascending=True).head(1)

                if not cheapest.empty:

                    c = cheapest.iloc[0]

                    st.success(f"Cheapest: \*\*{c['store']}\*\* — ₹{int(c['price'])} (qty {int(c['qty'])})")

                st.dataframe(df, use\_container\_width=True)

    with col2:

        st.info("Tip: pick a Category in the sidebar to narrow the store set used for comparison.")

# -------------------- Tab 3: Shopping List --------------------

with tabs[2]:

    st.subheader("Smart Shopping List + Budget Mode")

    items = st.text\_area("Enter items (comma-separated)", value="Rice 10kg, XYZ Shampoo, Milk Lotion")

    if st.button("Optimize Shopping List"):

        res = ui\_shopping\_list(items, city, category or None, radius)

        if "message" in res:

            st.warning(res["message"])

        else:

            if res["plan"]:

                for p in res["plan"]:

                    with st.container():

                        st.markdown(f"\*\*Store #{p['store\_id']}\*\* — Subtotal: ₹{int(p['subtotal'])}")

                        lines = [f"- {k}: ₹{int(v)}" for k, v in p["items"].items()]

                        st.markdown("\n".join(lines))

                st.success(f"Total estimated cost: ₹{int(res['total\_cost'])}")

            if res["not\_found"]:

                st.error(f"Not found: {', '.join(res['not\_found'])}")

            if res["covered\_all"]:

                st.caption("✅ All items covered")

# -------------------- Tab 4: Alerts (Demo) --------------------

with tabs[3]:

    st.subheader("Real-Time Stock & Deal Alerts (Demo)")

    a1, a2 = st.columns(2)

    with a1:

        sub\_prod = st.text\_input("Subscribe for product", value="XYZ Shampoo")

        sub\_city = st.text\_input("City for alerts", value=city)

        if st.button("Subscribe"):

            st.success(subscribe\_alert(sub\_prod, sub\_city)["message"])

    with a2:

        sim\_prod = st.text\_input("Simulate restock for product", value="XYZ Shampoo")

        sim\_store = st.number\_input("Store ID to restock", value=1, min\_value=1, step=1)

        if st.button("Simulate Restock"):

            st.success(simulate\_restock(sim\_prod, int(sim\_store))["message"])

    st.caption("Note: Alerts are in-memory for demo. Use Firestore for persistence.")

# -------------------- Tab 5: Chatbot --------------------

with tabs[4]:

    st.subheader("Voice + AI Chatbot Assistant")

    st.caption("Examples:\n- Where can I get a laptop bag under ₹1000 near me?\n- Find an open pharmacy in Hyderabad within 4 km.\n- Optimize this list: rice 10kg, xyz shampoo, milk lotion")

    query = st.text\_input("Your message")

    if st.button("Ask 🤖"):

        if query.strip():

            with st.spinner("Thinking..."):

                st.success(run\_agent(query))

        else:

            st.warning("Type something first 🙂")

# ------------------------------ end app.py ------------------------------

#####

# Ask for ngrok token once (if you don't have one, create at dashboard.ngrok.com)

from getpass import getpass

import subprocess, time

from pyngrok import ngrok

# Optional but recommended: paste token to get a public URL

NGROK\_TOKEN = getpass("Paste your ngrok authtoken (or press Enter to skip): ")

if NGROK\_TOKEN:

    # Configure ngrok with your token

    !ngrok config add-authtoken {NGROK\_TOKEN}

# Kill any previous Streamlit processes (ignore errors)

!pkill -f "streamlit run app.py" 2>/dev/null || true

# Start Streamlit on port 8501

proc = subprocess.Popen(["streamlit", "run", "app.py", "--server.port", "8501"])

# Give it a moment to boot

time.sleep(3)

# Create a tunnel to the port (works even without token but may be limited)

public\_url = ngrok.connect(8501)

print("🌐 Public App URL:", public\_url)

print("If the URL shows a 502 initially, wait ~5s and refresh.")

apis:  
 **Geocoding API**

 **Places API**

 **Distance Matrix API**

 **Directions API**