Prakhar Jha Weather App — Runnable Starter Repo

This document contains a complete, runnable starter repo for the weather app described in your brief. It uses **FastAPI**, **PostgreSQL** (docker-compose), **SQLAlchemy**, **geopy (Nominatim)** for geocoding and **OpenWeatherMap** for weather. A minimal frontend `index.html` demonstrates search, candidate selection, geolocation and weather display.

> **Important:** Replace `OPENWEATHER_API_KEY` in `.env` with your key. Nominatim is used for geocoding — cache responses for production.

File tree

```
prakhar-weather-starter/
├─ .env.example
├─ Dockerfile
├─ docker-compose.yml
├─ requirements.txt
├─ README.md
├─ app/
│  ├─ main.py
│  ├─ database.py
│  ├─ models.py
│  ├─ schemas.py
│  ├─ crud.py
│  ├─ geocode.py
│  ├─ weather_client.py
│  ├─ routers.py
│  └─ static/
│     └─ index.html
└─ alembic/  (optional if you want migrations)
```

.env.example

```
DATABASE_URL=postgresql://postgres:postgres@db:5432/weatherdb
OPENWEATHER_API_KEY=YOUR_OPENWEATHER_KEY
APP_HOST=0.0.0.0
APP_PORT=8000
```

---

requirements.txt

```
fastapi
uvicorn[standard]
sqlalchemy
psycopg2-binary
httpx
pydantic
python-dotenv
geopy
pandas
python-multipart
reportlab
```

---

docker-compose.yml

```yaml
version: '3.8'
services:
  db:
    image: postgres:15
    environment:
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: weatherdb
    ports:
      - "5432:5432"
    volumes:
      - dbdata:/var/lib/postgresql/data
  web:
    build: .
    command: uvicorn app.main:app --host 0.0.0.0 --port 8000 --reload
    ports:
      - "8000:8000"
    env_file: .env
    depends_on:
      - db
volumes:
  dbdata:
```

Dockerfile

```dockerfile
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
```

```
COPY . .
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

---

app/database.py

```python
# app/database.py
import os
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from dotenv import load_dotenv

load_dotenv()
DATABASE_URL = os.getenv('DATABASE_URL')

engine = create_engine(DATABASE_URL, echo=False)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
```

---

app/models.py

```python
# app/models.py
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer, String, Float, DateTime, JSON, Text
from datetime import datetime

Base = declarative_base()

class SavedRequest(Base):
    __tablename__ = 'saved_requests'
    id = Column(Integer, primary_key=True, index=True)
    location_input = Column(String, nullable=False)
    resolved_name = Column(String, nullable=True)
    latitude = Column(Float, nullable=False)
    longitude = Column(Float, nullable=False)
    date_from = Column(DateTime, nullable=False)
    date_to = Column(DateTime, nullable=False)
    snapshot = Column(JSON, nullable=True)
    notes = Column(Text, nullable=True)
    created_at = Column(DateTime, default=datetime.utcnow)
```

---

app/schemas.py

```python
# app/schemas.py
from pydantic import BaseModel, validator
from datetime import datetime
from typing import Optional

class LocationQuery(BaseModel):
    query: str

class CreateSavedRequest(BaseModel):
    location_input: str
    date_from: datetime
    date_to: datetime
    notes: Optional[str] = None

    @validator('date_to')
    def validate_dates(cls, v, values):
        if 'date_from' in values and v < values['date_from']:
            raise ValueError('date_to must be after date_from')
        return v

class SavedRequestOut(BaseModel):
    id: int
    location_input: str
    resolved_name: Optional[str]
    latitude: float
    longitude: float
    date_from: datetime
    date_to: datetime
    snapshot: Optional[dict]
    notes: Optional[str]
    created_at: datetime

    class Config:
        orm_mode = True
```

---

app/geocode.py

```python
# app/geocode.py
from geopy.geocoders import Nominatim
from geopy.exc import GeocoderTimedOut

geolocator = Nominatim(user_agent='prakhar_weather_app', timeout=10)

def geocode(query: str, limit: int = 3):
    try:
        locs = geolocator.geocode(query, exactly_one=False, limit=limit)
    except GeocoderTimedOut:
        return []
    if not locs:
        return []
    results = []
    for l in locs:
        results.append({
            'name': l.address,
            'lat': l.latitude,
            'lon': l.longitude,
            'raw': l.raw
        })
    return results

# simple reverse helper (not used in core but handy)

def reverse(lat, lon):
    try:
        loc = geolocator.reverse((lat, lon))
    except GeocoderTimedOut:
        return None
    if not loc:
        return None
    return {'name': loc.address, 'lat': loc.latitude, 'lon': loc.longitude,
'raw': loc.raw}
```

---

app/weather_client.py

```
# app/weather_client.py
import os
import httpx
from datetime import datetime
from dotenv import load_dotenv

load_dotenv()
API_KEY = os.getenv('OPENWEATHER_API_KEY')
BASE = 'https://api.openweathermap.org/data/2.5'

async def get_current_weather(lat: float, lon: float):
    url = f"{BASE}/weather"
    params = {'lat': lat, 'lon': lon, 'appid': API_KEY, 'units': 'metric'}
    async with httpx.AsyncClient(timeout=15) as client:
        r = await client.get(url, params=params)
        r.raise_for_status()
        return r.json()

async def get_5day_forecast(lat: float, lon: float):
```

```python
    url = f"{BASE}/forecast"
    params = {'lat': lat, 'lon': lon, 'appid': API_KEY, 'units': 'metric'}
    async with httpx.AsyncClient(timeout=20) as client:
        r = await client.get(url, params=params)
        r.raise_for_status()
        return r.json()

# Aggregate 3-hourly forecast to daily summary

def aggregate_daily(forecast_json):
    days = {}
    for item in forecast_json.get('list', []):
        dt = datetime.utcfromtimestamp(item['dt'])
        key = dt.date().isoformat()
        days.setdefault(key, {'temps': [], 'weathers': []})
        days[key]['temps'].append(item['main']['temp'])
        days[key]['weathers'].append(item['weather'][0])
    out = []
    for d, data in days.items():
        out.append({
            'date': d,
            'temp_min': min(data['temps']),
            'temp_max': max(data['temps']),
            'weather_sample': data['weathers'][len(data['weathers'])//2]
        })
    return out
```

app/crud.py

```python
# app/crud.py
from sqlalchemy.orm import Session
from .models import SavedRequest

def create_saved_request(db: Session, *, payload, snapshot):
    sr = SavedRequest(
        location_input=payload.location_input,
        resolved_name=getattr(payload, 'resolved_name', None),
        latitude=payload.latitude,
        longitude=payload.longitude,
        date_from=payload.date_from,
        date_to=payload.date_to,
        snapshot=snapshot,
        notes=payload.notes,
    )
    db.add(sr)
    db.commit()
    db.refresh(sr)
    return sr
```

```python
def get_saved_requests(db: Session, skip: int = 0, limit: int = 100):
    return
db.query(SavedRequest).order_by(SavedRequest.created_at.desc()).offset(skip).limit(limit).all(

def get_saved_request(db: Session, id: int):
    return db.query(SavedRequest).filter(SavedRequest.id==id).first()

def update_saved_request(db: Session, id: int, updates: dict):
    sr = get_saved_request(db, id)
    if not sr:
        return None
    for k, v in updates.items():
        if hasattr(sr, k):
            setattr(sr, k, v)
    db.commit()
    db.refresh(sr)
    return sr

def delete_saved_request(db: Session, id: int):
    sr = get_saved_request(db, id)
    if not sr:
        return False
    db.delete(sr)
    db.commit()
    return True
```

---

app/routers.py

```python
# app/routers.py
from fastapi import APIRouter, HTTPException, Depends
from sqlalchemy.orm import Session
from .schemas import LocationQuery, CreateSavedRequest, SavedRequestOut
from .geocode import geocode
from .weather_client import get_current_weather, get_5day_forecast,
aggregate_daily
from .crud import create_saved_request, get_saved_requests,
get_saved_request, update_saved_request, delete_saved_request
import asyncio
from .database import SessionLocal

router = APIRouter()

# DB dependency
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

@router.post('/geocode')
async def api_geocode(payload: LocationQuery):
    candidates = geocode(payload.query)
    if not candidates:
        raise HTTPException(status_code=404, detail='Location not found')
    return {'candidates': candidates}

@router.get('/weather')
async def api_weather(lat: float, lon: float):
    try:
        current, forecast = await asyncio.gather(
            get_current_weather(lat, lon),
            get_5day_forecast(lat, lon)
        )
    except Exception as e:
        raise HTTPException(status_code=502, detail=str(e))
    daily = aggregate_daily(forecast)
    return {'current': current, 'forecast_daily': daily}

@router.post('/saved', response_model=SavedRequestOut)
async def api_create_saved(req: CreateSavedRequest, db: Session =
Depends(get_db)):
    # geocode
    candidates = geocode(req.location_input, limit=1)
    if not candidates:
        raise HTTPException(status_code=404, detail='Location not found')
    loc = candidates[0]
    current = await get_current_weather(loc['lat'], loc['lon'])
    forecast = await get_5day_forecast(loc['lat'], loc['lon'])
    snapshot = {'current': current, 'forecast': forecast}
    payload = type('obj',(object,),{
        'location_input': req.location_input,
        'resolved_name': loc['name'],
        'latitude': loc['lat'],
        'longitude': loc['lon'],
        'date_from': req.date_from,
        'date_to': req.date_to,
        'notes': req.notes
    })
    sr = create_saved_request(db, payload=payload, snapshot=snapshot)
    return sr

@router.get('/saved')
def api_list_saved(skip: int = 0, limit: int = 100, db: Session =
Depends(get_db)):
    return get_saved_requests(db, skip=skip, limit=limit)

@router.get('/saved/{id}', response_model=SavedRequestOut)
```

```python
def api_get_saved(id: int, db: Session = Depends(get_db)):
    sr = get_saved_request(db, id)
    if not sr:
        raise HTTPException(status_code=404, detail='Not found')
    return sr

@router.put('/saved/{id}', response_model=SavedRequestOut)
def api_update_saved(id: int, updates: dict, db: Session = Depends(get_db)):
    if 'date_from' in updates and 'date_to' in updates and updates['date_to']
< updates['date_from']:
        raise HTTPException(status_code=400, detail='Invalid date range')
    sr = update_saved_request(db, id, updates)
    if not sr:
        raise HTTPException(status_code=404, detail='Not found')
    return sr

@router.delete('/saved/{id}')
def api_delete_saved(id: int, db: Session = Depends(get_db)):
    ok = delete_saved_request(db, id)
    if not ok:
        raise HTTPException(status_code=404, detail='Not found')
    return {'ok': True}
```

---

app/main.py

```python
# app/main.py
from fastapi import FastAPI
from .routers import router
from .database import engine
from . import models
from fastapi.staticfiles import StaticFiles

models.Base.metadata.create_all(bind=engine)
app = FastAPI(title='Prakhar Jha Weather App')
app.include_router(router)
app.mount('/static', StaticFiles(directory='app/static'), name='static')
```

---

app/static/index.html

```html
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>Prakhar Jha Weather</title>
  <style>body{font-family:Arial;padding:16px} .weather{border:1px solid
```

```
#ddd;padding:12px;margin-top:8px}</style>
</head>
<body>
  <h1>Prakhar Jha Weather</h1>
  <button id="infoBtn">ℹ️ Info</button>
  <div>
    <input id="locInput" placeholder="Enter city, zipcode, coords or
landmark" style="width:60%" />
    <button id="searchBtn">Search</button>
    <button id="gpsBtn">Use my location</button>
  </div>
  <div id="candidates"></div>
  <div id="result"></div>

  <script>
    const apiBase = location.origin; // assumes same origin

    document.getElementById('searchBtn').onclick = async () => {
      const q = document.getElementById('locInput').value;
      const res = await fetch(`${apiBase}/geocode`, {
        method:'POST', headers:{'Content-Type':'application/json'},
        body: JSON.stringify({query: q})
      });
      if(!res.ok){ alert('No location found'); return; }
      const data = await res.json();
      const container = document.getElementById('candidates');
      container.innerHTML = '';
      data.candidates.forEach(c => {
        const btn = document.createElement('button');
        btn.textContent = c.name;
        btn.onclick = () => fetchWeather(c.lat, c.lon, c.name);
        container.appendChild(btn);
      });
    };

    async function fetchWeather(lat, lon, name){
      const res = await fetch(`${apiBase}/weather?lat=${lat}&lon=${lon}`);
      const data = await res.json();
      const r = document.getElementById('result');
      r.innerHTML = `<h3>${name}</h3>` +
        `<div class='weather'><h4>Now: $
{data.current.weather[0].description}, ${data.current.main.temp} °C</h4>` +
        `<img src='http://openweathermap.org/img/wn/$
{data.current.weather[0].icon}@2x.png' /></div>` +
        `<div><h4>5-day summary</h4>` + data.forecast_daily.map(d => `<div>$
{d.date}: ${d.temp_min} - ${d.temp_max}°C - ${d.weather_sample.description}</
div>`).join('') + `</div>`;
    }

    document.getElementById('gpsBtn').onclick = () => {
      navigator.geolocation.getCurrentPosition(pos => {
        const lat = pos.coords.latitude;
        const lon = pos.coords.longitude;
        fetchWeather(lat, lon, `Your location (${lat.toFixed(2)},$
{lon.toFixed(2)})`);
      }, err => alert('Geolocation failed: ' + err.message));
    };

    document.getElementById('infoBtn').onclick = () => {
      window.open('https://www.linkedin.com/company/product-manager-
accelerator', '_blank');
    };
  </script>
</body>
</html>
```

README.md (short)

~~~
# Prakhar Jha Weather App — Starter

## Quick start
1. Copy `.env.example` to `.env` and fill `OPENWEATHER_API_KEY`.
2. Start with Docker:
   ```bash
   docker-compose up --build
   ```
3. Open http://localhost:8000/static/index.html

API endpoints live at http://localhost:8000 (see `app/routers.py`).

Notes:
- Historical bulk data requires a paid weather API. This starter stores
current snapshot + 5-day forecast for saved records.
- For production: add caching, rate-limiting, and authentication.
~~~

What I didn't include (but recommended)

- Alembic migration files (you can generate with `alembic init`)
- Unit tests (pytest) and mocking of external APIs
- Advanced export endpoints (CSV/PDF) — easy to add using `pandas` and `reportlab`

If you want, I can now: - Generate this repo as downloadable files (I can paste every file in chat or package scripts), or - Add Alembic migration steps and sample pytest tests, or - Replace PostgreSQL with SQLite for a single-file demo for quick local testing.

Tell me which of the above you'd like **now** and I will produce it immediately.