Para este proyecto enfocado en FamaEmpleo, la idea central es construir un backend que permita extraer información de las ofertas de manera estructurada, almacenarla y exponerla para su análisis o visualización sin depender de un frontend sofisticado. Se utilizará Python como lenguaje principal, apoyado en Requests y BeautifulSoup para el scraping inicial, considerando que gran parte del contenido puede estar en HTML estático; Playwright se contempla como alternativa si se detecta contenido dinámico o la necesidad de simular navegación más compleja, incluyendo login eventual. Los datos se almacenarán en PostgreSQL, creando tablas que representen las ofertas, las búsquedas realizadas y posibles notas o comentarios asociados, de manera que cada registro pueda identificarse por fuente y fecha de ejecución, permitiendo luego filtrado y consultas eficientes. Flask actuará como API central, recibiendo las solicitudes del frontend o de pruebas locales y entregando la información de forma uniforme, mientras que los scrapers se desarrollan como módulos independientes que consultan la base de datos y transforman los resultados según los filtros definidos, manteniendo un flujo modular y escalable para agregar nuevas fuentes en el futuro. La arquitectura general contempla un archivo principal para la API, un módulo de modelos que define la estructura de la base de datos y un scraper específico de FamaEmpleo, con la posibilidad de automatizar su ejecución mediante un scheduler local; el objetivo es mantener la operación íntegramente en desarrollo local, asegurando que las librerías y dependencias estén claramente definidas y que los procesos de scraping, almacenamiento y exposición de datos sigan un flujo coherente y repetible.

**Propuesta de backend para proyecto Scraper FamaEmpleo**

**1️⃣ Stack recomendado**

| **Componente** | **Tecnología** | **Rol / Función** |
| --- | --- | --- |
| Lenguaje | **Python 3.x** | Automatización, scraping, backend API |
| Automatización / Scraping | **Requests + BeautifulSoup** (primera opción)  **Playwright** (opción para JS dinámico) | Requests + BS: páginas simples, HTML estático  Playwright: páginas que cargan contenido dinámico vía JavaScript, navegación más robusta, login simulado si se requiere |
| Base de datos | **PostgreSQL** | Almacenamiento estructurado de ofertas y búsquedas |
| ORM / Migraciones | **SQLAlchemy + Alembic / Flask-Migrate** | Definición de modelos y control de migraciones |
| Backend / API | **Flask** | Exposición de endpoints para frontend / filtrado / gestión de datos |
| Scheduler (opcional) | **cron (Linux) o Task Scheduler (Windows)** | Ejecutar scraping automáticamente, guardar resultados diarios |

Nota: Todo se mantiene **en local** para desarrollo y pruebas iniciales.

**2️⃣ Base de datos**

Se recomienda una sola base de datos central para mantener la escalabilidad y simplicidad, con **tablas modulares**:

**Tabla: ofertas**

| **Campo** | **Tipo** | **Descripción** |
| --- | --- | --- |
| id | SERIAL PRIMARY KEY | Identificador interno |
| titulo | VARCHAR | Título del puesto |
| empresa | VARCHAR | Nombre de la empresa |
| ubicacion | VARCHAR | Ciudad / departamento |
| fecha\_publicacion | DATE | Fecha de publicación |
| url | VARCHAR | Enlace directo a la oferta |
| descripcion | TEXT | Texto completo de la oferta |
| fuente | VARCHAR | Ej: "FamaEmpleo" (permite escalar a otras páginas) |

**Tabla: busquedas**

| **Campo** | **Tipo** | **Descripción** |
| --- | --- | --- |
| id | SERIAL PRIMARY KEY | Identificador de búsqueda |
| termino | VARCHAR | Palabra clave buscada |
| filtros | JSON | Parámetros de búsqueda (ubicación, categoría, salario) |
| fecha\_ejecucion | TIMESTAMP | Registro de cuándo se ejecutó |

**Tabla: notas (opcional)**

| **Campo** | **Tipo** | **Descripción** |
| --- | --- | --- |
| id | SERIAL PRIMARY KEY | Identificador de nota |
| oferta\_id | INTEGER | FK a ofertas |
| nota | TEXT | Comentario personal o etiqueta |

**Concepto clave:** la tabla fuente permite un backend modular y escalable cuando agreguemos otros scrapers.

**3️⃣ Flujo backend / scraping**

**Opción A: Scraping simple (HTML estático)**

1. Requests obtiene HTML de la búsqueda en FamaEmpleo.
2. BeautifulSoup parsea las ofertas (título, empresa, ubicación, descripción).
3. Los datos se guardan en PostgreSQL.
4. Flask expone endpoints: /ofertas, /busquedas, /notas.
5. Frontend (React) consume la API para mostrar datos filtrados.

Ventaja: simple, rápido y bajo riesgo. No requiere abrir navegador.

**Opción B: Scraping con Playwright (JS dinámico / login)**

1. Playwright abre un navegador simulado (headless o visible).
2. Inicia sesión si se requiere, usando variables de entorno para usuario/contraseña.
3. Navega a la búsqueda, aplica filtros y espera a que se cargue contenido dinámico.
4. Extrae datos (igual que en Opción A).
5. Guarda en PostgreSQL.
6. Flask expone API para frontend.

Ventaja: maneja páginas dinámicas, navegación compleja y login.  
Desventaja: más pesado, requiere instalación de dependencias (Playwright + drivers).

**4️⃣ Dependencias / instalación inicial (Python)**

# Entorno virtual

python -m venv venv

source venv/bin/activate # Linux/Mac

venv\Scripts\activate # Windows

# Instalación básica

pip install flask sqlalchemy psycopg2-binary alembic flask-migrate requests beautifulsoup4

# Si optamos por Playwright

pip install playwright

playwright install

**5️⃣ Arquitectura inicial**

/fama\_scraper\_project

│

├─ /scrapers

│ ├─ fama\_empleo.py # Scraper específico para FamaEmpleo

│

├─ /backend

│ ├─ app.py # Flask app

│ ├─ models.py # SQLAlchemy models

│ ├─ routes.py # Endpoints API

│ ├─ database.py # Conexión a PostgreSQL

│

├─ /config

│ ├─ config.py # Configuración de DB, variables de entorno

│

├─ /migrations

│ └─ ... # Alembic migraciones

│

└─ requirements.txt # Dependencias del proyecto

**6️⃣ Consideraciones clave**

* **Filtros y búsquedas:** Guardarlos en tabla busquedas permite reutilizarlos y automatizar ejecuciones diarias.
* **Login:** Por ahora opcional; si luego se requiere, Playwright es la opción más segura y flexible.
* **Simulación / pruebas:** Scraper puede ejecutarse sin guardar datos para validar flujo.
* **Frontend:** Puede ser React simple que consuma /ofertas y permita filtrar por fecha, ubicación, empresa.

**7️⃣ Primeros pasos recomendados**

1. Crear entorno virtual y base de datos PostgreSQL.
2. Crear tablas ofertas y busquedas con SQLAlchemy + Alembic.
3. Implementar scraper básico con Requests + BeautifulSoup para FamaEmpleo (HTML estático).
4. Implementar Flask API simple para exponer /ofertas.
5. Validar flujo completo: scraper → DB → API → Frontend mínimo (tabla de datos).
6. Evaluar necesidad de Playwright si hay contenido dinámico o login obligatorio.

¡Exactamente! Ahora sí estamos entrando en la **lógica de negocio**, en la “salsa” del proyecto. Hasta aquí lo que hicimos fue preparar el **esqueleto del backend**: levantar Flask, configurar la base de datos, inicializar extensiones, definir modelos y exponer endpoints básicos. Todo eso ya es infraestructura, listo para soportar la lógica real del scraper y la manipulación de datos.

Ahora el siguiente paso es **planificar cómo va a funcionar el scraper de FamaEmpleo**, qué librerías vamos a usar, cómo vamos a organizar el código y cómo vamos a interactuar con nuestra API y la base de datos. No vamos a entrar todavía a escribir código línea por línea, sino a definir el flujo completo, los módulos y las decisiones estratégicas.

**Plan general de la lógica del scraper**

1. **Elección de la librería**
   * **Playwright**: Ideal si la web usa **JavaScript dinámico**. Permite navegar, hacer login y esperar a que la página cargue contenido antes de extraerlo.
   * **Requests + BeautifulSoup**: Útil si la página es **estática** y puedes obtener los datos directamente desde HTML. Más rápido y liviano, pero no sirve con JS dinámico.
   * **Decision**: FamaEmpleo parece tener **contenido cargado dinámicamente**, por lo que **Playwright** es la opción más robusta. Requests/BS puede servir como prueba o para páginas simples.
2. **Organización del módulo scraper**
   * Crear carpeta app/scrapers/fama (modular por portal).
   * Dentro, un archivo scraper.py que contenga funciones para:
     + Inicializar el navegador (Playwright)
     + Hacer login si es necesario
     + Realizar búsquedas según los términos y filtros que guardamos en Busquedas
     + Extraer datos de cada oferta (titulo, empresa, ubicación, fecha, url, descripción)
     + Devolver los resultados en formato JSON o diccionario listo para guardar en Ofertas
3. **Integración con la base de datos**
   * Cada vez que el scraper obtenga ofertas, se conecta a **SQLAlchemy** y hace un db.session.add\_all() o inserciones individuales para la tabla Ofertas.
   * Se puede agregar lógica de **evitar duplicados**, por ejemplo comparando url o titulo + empresa.
4. **Gestión de filtros y búsquedas**
   * Usar la tabla Busquedas para leer los términos y filtros activos.
   * El scraper recorre cada búsqueda y ejecuta la consulta en la web, guardando resultados asociados a esa búsqueda si se desea (relacionando oferta\_id y busqueda\_id).
5. **Notas y seguimiento**
   * Endpoints /notas permiten agregar anotaciones manuales sobre ofertas.
   * Eventualmente, el scraper podría generar notas automáticas (por ejemplo, “Oferta ya existe” o “Nueva oferta agregada hoy”).
6. **Arquitectura general**
   * **Backend principal**: Flask + API + DB + extensiones
   * **Módulo scraper**: independiente, que se puede ejecutar como script, cron job local o comando Flask
   * **Integración**: el scraper llama a funciones que hacen db.session.add() → la API ya puede consultar los datos vía /api/ofertas
   * **Escalabilidad**: cada portal (FamaEmpleo, CompuTrabajo, etc.) tiene su propio módulo, con misma interfaz de entrada/salida.

💡 **Resumen de decisiones estratégicas:**

* Librería principal: **Playwright** (por JS dinámico)
* Scraper modular por portal → app/scrapers/<portal>
* Extraer datos → transformar a diccionario → guardar en base de datos
* Usar la tabla Busquedas como motor de queries para automatizar búsqueda de ofertas
* La API queda como **capa de consulta y visualización**, independiente del scraper

Si quieres, el siguiente paso sería **definir la estructura de archivos y funciones del scraper de FamaEmpleo**, indicando **qué hace cada función**, **qué devuelve** y **cómo se conecta con la base de datos**, todo en planificación, antes de escribir código.

¿Quieres que hagamos esa planificación del scraper ahora?