Arquitectura de Referencia: Motor de Búsqueda Federado y Contextual

Versión del Documento: 1.0 **Objetivo:** Definir la lógica, arquitectura y mejores prácticas para la implementación de un buscador empresarial inteligente.

---

1. Resumen Ejecutivo

El objetivo de este desarrollo no es simplemente filtrar una lista de textos, sino construir un **Orquestador de Búsqueda**. Este sistema debe ser capaz de interrogar múltiples fuentes de datos simultáneamente (Federated Search), normalizar la información heterogénea y entregarla al usuario enriquecida con contexto (relaciones inteligentes), garantizando una experiencia de usuario (UX) fluida y performante.

---

2. Arquitectura de Alto Nivel: Desacople Total

Para garantizar la escalabilidad y mantenibilidad, se recomienda estrictamente separar la lógica de obtención de datos (Backend) de la capa de presentación (Frontend).

El Backend como "Orquestador"

El Backend (Node.js/Express) actúa como una **Capa de Abstracción**. El Frontend **nunca** debe consultar directamente a la API externa ni al Bucket S3.

- **Ventaja:** Si mañana cambiamos S3 por Azure Blob Storage, o la API externa cambia sus credenciales, el Frontend no se entera ni se rompe. Solo se ajusta el Backend.

---

3. Lógica del Núcleo: Búsqueda Federada (Federated Search)

Este es el componente crítico cuando la información reside en silos diferentes (ej: Archivos PDF en un S3 y Datos de Inventario en una API de Terceros).

3.1 El Problema de la Secuencialidad

Un error común es realizar las búsquedas en fila (Waterfall):

1. Consultar S3 (Tarda 1s)
2. Esperar...
3. Consultar API (Tarda 1s)
4. **Tiempo Total:** 2 segundos. **(Inaceptable para UX)**

3.2 La Solución: Paralelismo (`Promise.all`)

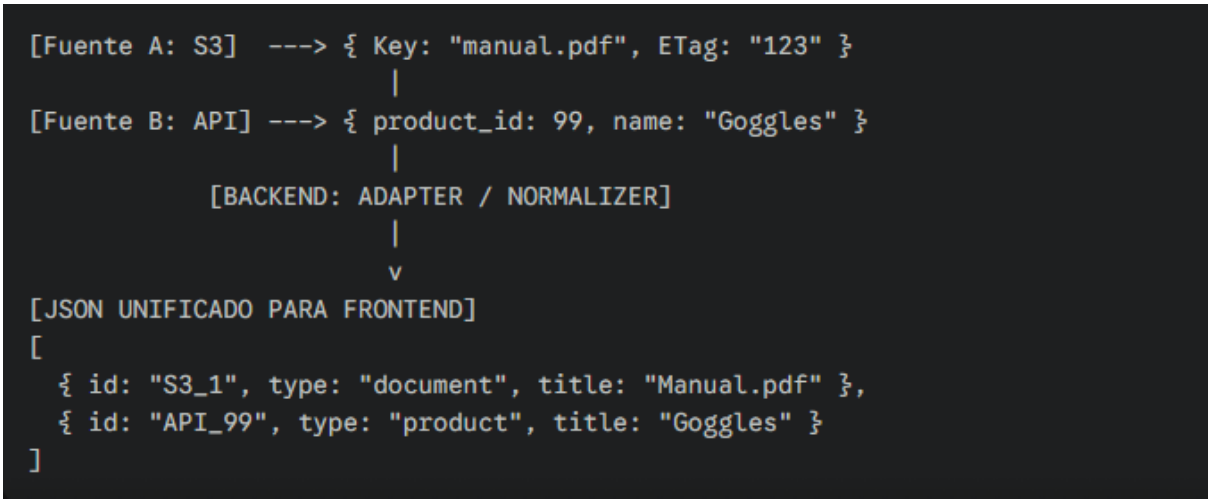
La arquitectura propuesta utiliza ejecución concurrente. Se disparan ambas peticiones al mismo instante.

- **Tiempo Total:** 1 segundo (El tiempo del servicio más lento, no la suma).

3.3 El Patrón "Adapter" (Normalización)

Dado que S3 devuelve metadatos de archivos (Key, Size) y la API devuelve datos de negocio (SKU, Stock), el Backend debe transformar ambos formatos a un **Modelo de Datos Unificado** antes de responder al Frontend.

Ejemplo de Flujo de Datos:



4. Lógica de Frontend (Agnóstica a Framework)

Independientemente de si se usa **React, Angular o HTML/JS**, estas son las lógicas que deben implementarse en el cliente:

4.1 Estrategia de "Debounce" (Anti-Saturación)

No se debe enviar una petición al servidor cada vez que el usuario presiona una tecla.

- **Lógica:** Esperar X milisegundos (ej. 300ms) desde que el usuario *deja* de escribir antes de lanzar la petición.
- **Beneficio:** Reduce la carga al servidor en un 90% y evita parpadeos en la interfaz.

4.2 Resaltado Inteligente (Smart Highlighting)

Para mejorar la legibilidad, el Frontend debe procesar el texto recibido.

- **Lógica:** Usar Expresiones Regulares (`RegExp`) para fragmentar el título/descripción basándose en el término de búsqueda y envolver las coincidencias en una etiqueta `<span>` con estilo (ej. fondo amarillo).
- **Nota:** Esto debe ser insensible a mayúsculas/minúsculas.

4.3 Manejo de Estado (Loading & Error)

La interfaz debe tener tres estados visuales claros:

1. **Idle/Results:** Muestra los datos.
2. **Loading:** Muestra un spinner o mensaje "Analizando fuentes externas...". Esto es vital en búsquedas federadas donde la red puede tardar.
3. **Error:** Si una de las fuentes falla, el sistema debe degradarse graciosamente (mostrar los resultados parciales de la fuente que sí respondió) o mostrar un mensaje amigable.

---

5. Optimizaciones de Rendimiento y UX

Para llevar el proyecto a nivel productivo, se recomiendan estas mejoras:

1. **Caché de Corta Duración (Redis):** Si la API de terceros es lenta, guardar la respuesta de una búsqueda popular (ej. "microscopio") en memoria (Redis) por 5-10 minutos. Las siguientes búsquedas serán instantáneas.
2. **Paginación o Lazy Loading:** Sí la búsqueda federada devuelve 500 resultados, no enviarlos todos al front. Enviar los primeros 20 y cargar más a demanda.
3. **Feedback Visual Inmediato:** Implementar "Skeleton Screens" (cajas grises parpadeantes) mientras carga la información para reducir la percepción de espera.

---

6. Plan de Pruebas (Test Cases)

Para certificar el éxito del desarrollo, el equipo de QA o los desarrolladores deben validar estos escenarios:

ID	Escenario	Prueba	Resultado Esperado
TC-01	**Búsqueda Unificada**	Buscar un término que exista en ambas fuentes (S3 y API).	La lista de resultados debe mostrar ítem mezclados de ambas fuentes, ordenados por relevancia.
TC-02	**Resiliencia (Fallo Parcial)**	Simular que la API externa está caída (Timeout) pero S3 funciona.	El sistema **NO** debe romperse. Debe mostrar los resultados de S3 y un aviso sutil: "Algunos sistemas no están disponibles".
TC-03	**Performance**	Buscar un término común.	El tiempo de respuesta no debe superar el tiempo de respuesta del servicio externo más lento.
TC-04	**Sanitización**	Buscar caracteres especiales (`<script>` , `%` , `/`).	El backend no debe crashear y el frontend no debe ejecutar código malicioso.
TC-05	**UX Highlighting**	Buscar "Lentes".	En la tarjeta de resultados, la palabra "Lentes" y "lentes" deben aparecer resaltadas.

7. Recomendación Tecnológica (Stack)

Basado en el éxito del prototipo, estas son las tecnologías recomendadas:

- **Backend: Node.js + Express**.
  - *Por qué:* Su manejo no bloqueante de I/O es ideal para esperar múltiples APIs externas al mismo tiempo sin "congelar" el servidor.
- **Frontend: React** (Implementado en prototipo) o **Angular**.
  - *Recomendación:* Si el equipo domina Angular, usar **RxJS** es ideal para manejar el *Debounce* y las llamadas HTTP, ya que es nativo del framework.
  - *Estilos:* Se recomienda CSS modular o utilitario para evitar conflictos de estilos globales.
- **Intercambio de Datos: JSON**.
  - Mantener la estructura estricta definida en el prototipo: `{ meta: {}, results: [], related: [] }`.