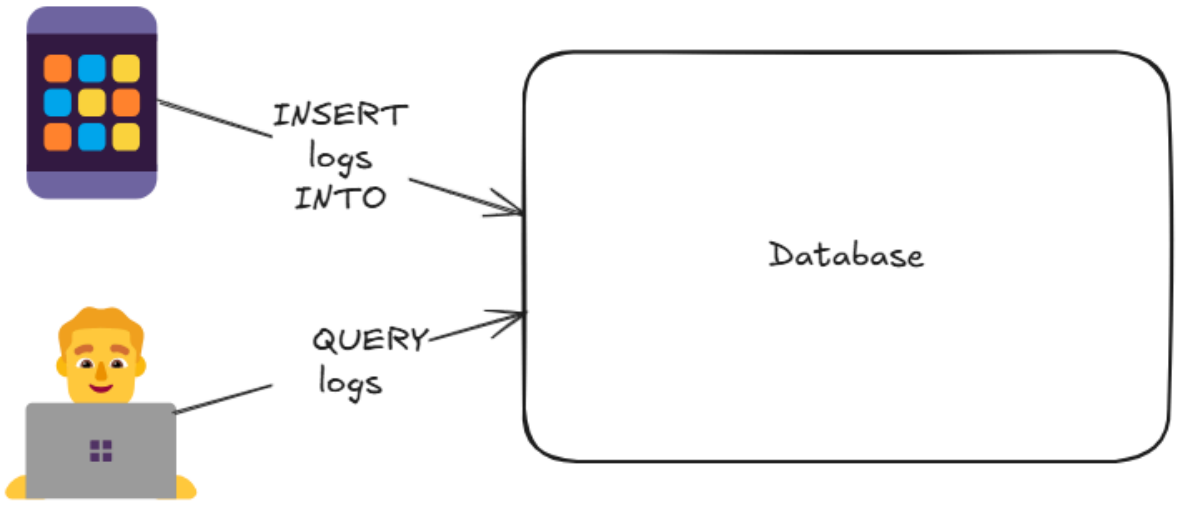
Desarrollemos nuestro “analizador” de logs

Ahora que ya vimos los conceptos principales, pongamos en juego todo lo aprendido en la clase y **desarrollemos el analizador de logs requerido por Umbrella Corp** ¡Vamos!

Primero que nada, entendamos el problema

En principio, la solución a este problema puede parecer sencilla ¿No? Puedes estar pensando en una solución de este estilo



Básicamente, se trata de usar una base de datos como fuente de almacenamiento de la información, donde cada uno de los sensores registre los datos de su sala y los usuarios puedan consultar la información almacenada ¡Bien pensado! Recuerda siempre comenzar por la solución más sencilla e ir agregando complejidad solo si el problema lo requiere.

Menos es más. Ahora bien, veamos por qué esta primera solución no será suficiente.

Principalmente, no lo será porque necesitamos realizar consultas de los últimos 5 minutos con tiempos de respuesta muy rápidos (baja latencia). Si consultamos

directamente desde una base de datos relacional, no podremos obtener los tiempos de respuesta bajos que esperamos ¿Por qué? Por dos razones principales:

1. **Recursos consumidos por las inserciones de datos:** La mayoría de los recursos estarán ocupados con la tarea de insertar la información enviada por los sensores. A medida que se sumen más sensores, se necesitarán más recursos para procesar las inserciones, lo que puede hacer que las consultas se vuelvan más lentas.

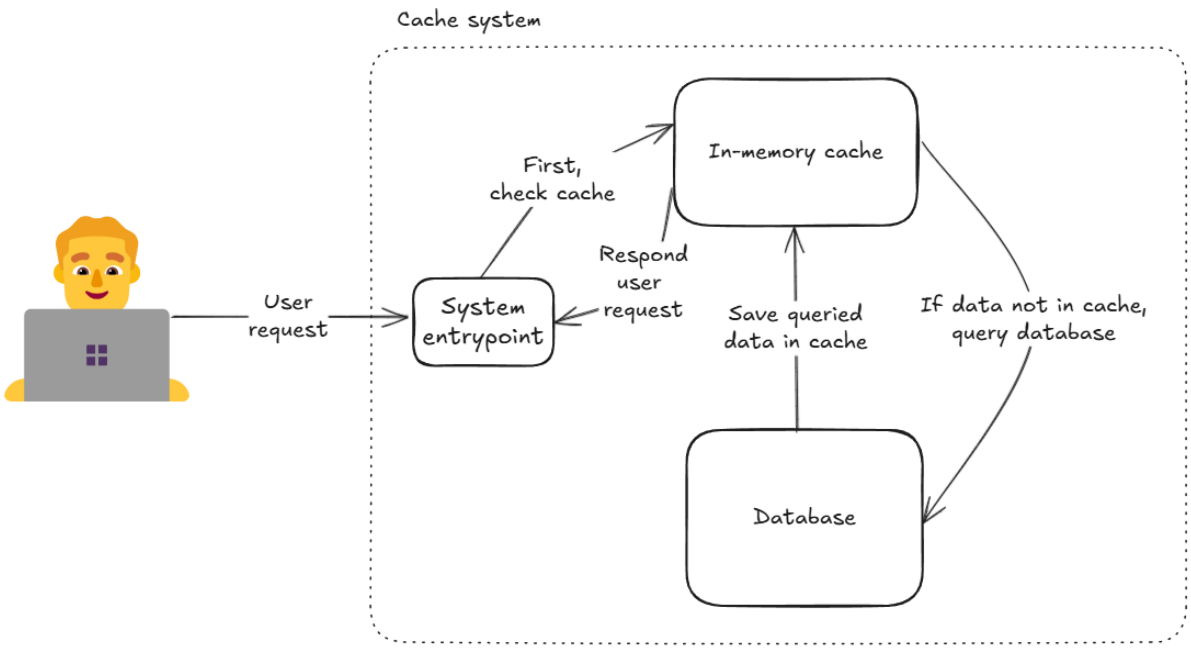
2. **Crecimiento exponencial de los logs:** Como la actualización de los logs ocurre cada segundo, el volumen de datos crecerá de forma exponencial. Con el tiempo, la base de datos se tardará cada vez más en consultar solo los últimos 5 minutos de datos, ya que tendrá que revisar una cantidad mucho mayor de registros. Además, internamente, la base de datos necesitará acceder al disco duro para obtener la información, lo que hará que la consulta sea aún más lenta.

Así pues, tendremos que buscar otra posible solución, que satisfaga el requerimiento de la baja latencia.

¡Nueva solución en marcha!

La solución para el problema de rendimiento es algo ampliamente usado en desarrollo backend: un caché. Pero... ¿Recuerdas qué es un caché?

Un **caché** es un sistema que almacena en **memoria de acceso rápido (RAM)** los datos que se han accedido recientemente o que son más frecuentemente solicitados, de manera que, en lugar de tener que hacer una consulta a la base de datos cada vez, se pueda responder mucho más rápido con los datos almacenados en el caché. Esto reduce significativamente el tiempo de respuesta y la carga sobre la base de datos. Una de las soluciones de caché más utilizadas en la industria es Redis. Un sistema con caché se ve de esta forma:



Funciona de la siguiente manera:

1. Primero, el sistema verifica si la información solicitada se encuentra disponible en el caché temporal (Temporal caché), el cual está diseñado para almacenar en memoria datos consultados recientemente.
    a. Si los datos se encuentran en el caché, se responden inmediatamente al usuario, lo que permite una respuesta rápida y con baja latencia.
    b. En caso de que la información no esté en el caché, el sistema procede a consultar la base de datos (Database). Una vez que obtiene la información desde la base de datos, esta se guarda en el caché para acelerar futuras consultas, y luego se envía al usuario como respuesta.

Ahora que hemos determinado que necesitamos un sistema de caché, surge una nueva pregunta: ¿cómo permitiremos que tanto los sensores como los usuarios consulten o almacenen información en este caché? Lo haremos a través de una API.

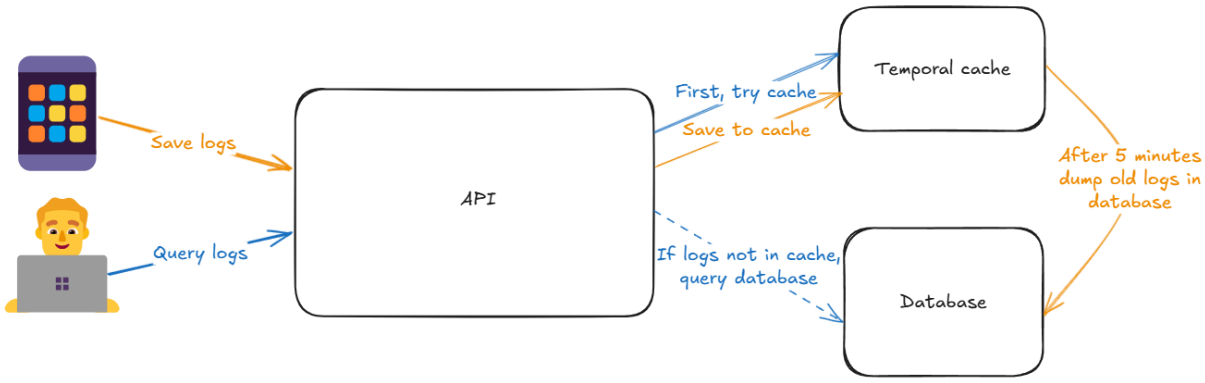
> 💡Reminder → Una **API (Application Programming Interface)** es una interfaz que permite que distintas aplicaciones se comuniquen entre sí, y en este caso, actúa como un punto central de acceso al sistema. Utiliza el protocolo HTTP, lo cual significa que puede ser accedida a través de internet. Las APIs exponen

endpoints, que son rutas específicas diseñadas para realizar distintas operaciones (como guardar o consultar datos).

Estas operaciones se realizan mediante **métodos HTTP**, como:

- **GET**, para consultar información.
- **POST**, para enviar o almacenar nuevos datos.
- **PUT/PATCH**, para actualizar información existente.
- **DELETE**, para eliminar registros.

En resumen, la API será la puerta de entrada tanto para los sensores que reportan datos, como para los usuarios que consultan el estado de las salas. Todo esto de forma estructurada, segura y rápida. Con esto en mente nuestra propuesta de solución se vería de esta forma:



1. **Los sensores** recopilan información de las salas y la envían a través de una **API**, usando métodos HTTP como `POST`, para almacenar los datos en una base de datos.
2. **La base de datos** actúa como almacenamiento central de toda la información histórica generada por los sensores.
3. Para consultas frecuentes, especialmente aquellas que requieren baja latencia (como obtener los logs de los últimos 5 minutos), el sistema utilizará un **caché temporal**.
4. Cuando un usuario (o sistema) realiza una solicitud de datos a través de la API:
    - Primero se intenta obtener la información desde el **caché**.
    - Si la información está en el caché, se responde rápidamente.
    - Si no está en el caché, se consulta la base de datos, se guarda el resultado en el caché, y luego se responde al usuario.

Ahora si, definamos la arquitectura de la solución

Hasta ahora vamos por buen camino, pero aún nos faltan definir algunos puntos clave para completar nuestra solución. Específicamente, debemos decidir:

- Qué **endpoints** va a tener nuestra API.
- Qué tecnología utilizaremos para nuestro **caché**, o si construiremos uno desde cero.
- Qué **base de datos** usaremos para almacenar la información de manera persistente.

Sobre los endpoints

Necesitaremos implementar tres endpoints principales:

- `POST /logs`: Para que los sensores puedan enviar y almacenar los logs.
- `GET /logs`: Para que los usuarios puedan consultar los logs en un rango de fechas (`start_date` y `end_date`).

- `GET /logs/all`: Para obtener todos los logs que estén actualmente en el caché (pensado para lecturas rápidas, como por ejemplo las del último intervalo de 5 minutos).

Sobre la base de datos

Para mantener la solución simple durante las primeras etapas de desarrollo, podemos utilizar **SQLite**, una base de datos ligera, fácil de integrar y sin necesidad de configuración externa. Sin embargo, si el sistema crece o se proyecta a producción, lo más recomendable sería usar una solución más robusta como **PostgreSQL** o **MySQL**, que permiten un mejor manejo de concurrencia, escalabilidad y seguridad.

Sobre el caché

Aquí es donde entra en juego uno de los componentes más importantes de nuestra arquitectura: **el sistema de caché**.

Los sistemas de caché tradicionales —como Redis, Memcached o similares— funcionan almacenando de manera temporal los datos que han sido consultados recientemente y que no se encontraban previamente en el caché. Esta estrategia es útil en muchos casos, pero **no se ajusta del todo a nuestra necesidad**.

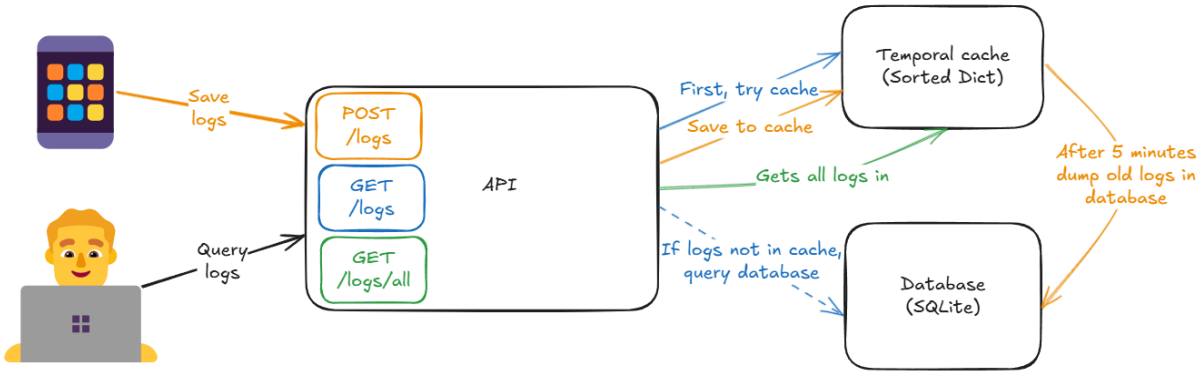
Nosotros no queremos almacenar simplemente lo que se consulta, sino mantener **únicamente los logs generados en los últimos 5 minutos**. Es decir, necesitamos un sistema que gestione activamente la ventana de tiempo que queremos conservar, eliminando cualquier dato que exceda ese umbral temporal.

Por esta razón, la mejor opción es **implementar nuestro propio sistema de caché en memoria**, diseñado específicamente para esta lógica. Esto nos dará control total sobre cómo se almacenan, expiran y se recuperan los datos, y asegurará que nuestras consultas sean rápidas y precisas dentro de esa ventana crítica de monitoreo.

🙂 *Además, no hay mejor forma de entender cómo funciona un sistema que construirlo uno mismo.*

Nuestra arquitectura final

Así pues, **dicho lo dicho y explicado lo explicado**, nuestra arquitectura propuesta quedaría de la siguiente manera:



Implementemos nuestro sistema de cache

Sabemos que nuestro sistema de caché debe ser extremadamente rápido en tiempo de acceso, y también hemos visto por qué conviene implementar nuestro propio servicio de caché.

¿Cómo lo implementamos?

Nuestro servicio de caché, tal y como se muestra en la figura anterior, cumple con dos funcionalidades principales:

1. **Almacenar los logs generados en los últimos 5 minutos**, permitiendo así consultas rápidas y eficientes sobre la información más reciente.

2. **Eliminar del caché y persistir en la base de datos aquellos logs que tengan más de 5 minutos de antigüedad**, garantizando que el caché se mantenga liviano y que la información histórica esté correctamente resguardada.

Así pues debemos de atacar y resolver estas dos funcionalidades.

Almacenar los logs generados en los últimos 5 minutos

Para el almacenamiento de logs, es fundamental utilizar una estructura de datos que permita un acceso eficiente, idealmente en tiempo constante **O(1)** o, en su defecto, en tiempo logarítmico **O(log n)**. Esto garantiza una baja latencia al consultar los datos, lo cual es esencial en sistemas que requieren respuestas rápidas.

Además, para simplificar la implementación, lo más conveniente es usar una estructura de tipo **clave-valor**. Al utilizar la **fecha y hora de creación del log como clave**, se facilita la búsqueda por rangos de tiempo, como por ejemplo, los últimos 5 minutos. Esta organización permite acceder a los datos de forma directa, manteniendo el rendimiento del sistema incluso con grandes volúmenes de información.

Ahora bien, también puede ocurrir que un log generado a las **10:21:01** llegue al sistema a las **10:22:43**. Este caso, conocido como **late arriving event**, plantea un reto adicional: aunque muchas estructuras no están ordenadas por defecto, el orden puede lograrse manualmente si es necesario. Sin embargo, esta operación implica un mayor costo computacional, por lo que debe considerarse cuidadosamente según los requisitos de precisión y rendimiento del sistema.

Estructura de datos	Descripción	Tiempo de acceso	Búsqueda por rango de fechas
`dict`	Diccionario o HashMap tradicional de Python.	Acceder a un valor por medio de una llave se hace en O(1). En el peor de los casos, donde hay múltiples valores para una misma llave (collision), se accede en O(n).	O(n)
`OrderedDict`	Diccionario que mantiene el orden de inserción de los elementos.	Acceder a un valor por medio de una llave se hace en O(1). En el peor de los casos, donde hay múltiples valores para una misma llave (collision), se accede en O(n).	O(n)
`SortedDict`	Diccionario que ordena automáticamente los	Acceder a un valor por medio de una llave se hace en O(log	O(log n)

	elementos por clave.	n).	

Dado el rendimiento consistente en tiempo logarítmico O(logn) tanto para el acceso por clave como para la búsqueda por rangos de fechas, `SortedDict` se presenta como una excelente opción para almacenar los logs en memoria. A diferencia de `dict` u `OrderedDict`, que ofrecen acceso en tiempo constante O(1) pero no están optimizados para búsquedas por rangos, `SortedDict` permite mantener las claves ordenadas, lo que facilita enormemente este tipo de consultas.

Además, en nuestro caso es muy probable que se generen múltiples logs con la misma marca de tiempo (por ejemplo, varios sensores reportando en el mismo segundo). Este escenario de "colisiones" puede deteriorar el rendimiento de estructuras basadas en hash como `dict`, mientras que con `SortedDict`, podemos almacenar listas de eventos por cada timestamp sin afectar el tiempo de búsqueda de la clave.

Eliminar del cache los logs “antiguos”

Para eliminar los logs antiguos del caché, necesitamos una estructura de datos que recorra los timestamps almacenados y conserve sólo los correspondientes a los últimos 5 minutos. La estructura de datos ideal para esta tarea debe garantizar bajos tiempos de acceso, inserción y eliminación tanto en la cabeza (primer elemento) como en la cola (último elemento), ya que será necesario insertar nuevos rangos temporales de manera constante, acceder a ellos para verificar si debemos mover la ventana de tiempo y eliminar los extremos para ajustar la ventana.

Estructura de datos	Tiempo de acceso	Tiempo de inserción	Tiempo de eliminación
Deque (Double-Ended Queue)	Tanto acceder al primer elemento (head), como el último (tail) se hace en O(1).	Tanto insertar el primer elemento (head), como el último (tail) se hace en O(1).	Tanto eliminar el primer elemento (head), como el último (tail) se hace en O(1).
Heap	Tanto acceder al primer elemento (head), como el último (tail) se hace en O(1).	Tanto insertar el primer elemento (head), como el último (tail) se hace en O(log n).	Mientras que eliminar el primer elemento se realiza en O(log n), eliminar el último elemento se hace en O(1).

Nuevamente, dado el rendimiento en tiempo constante dado por deque para este caso, esta será la estructura escogida para poner en marcha la ventana temporal.

Así funcionará el sistema de ventana temporal:

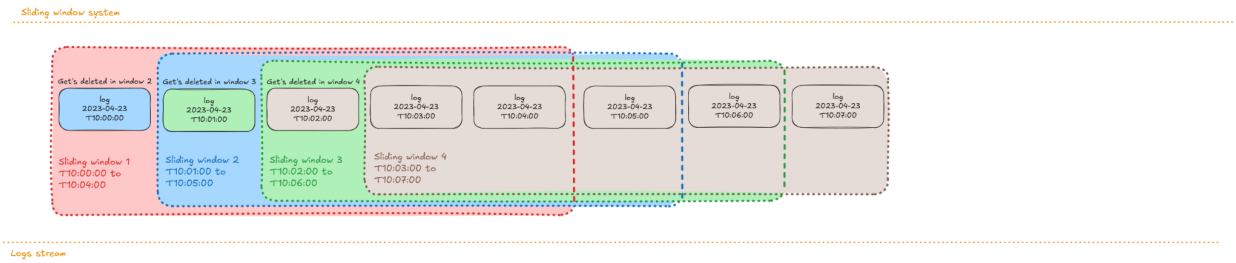
Funcionamiento del sistema de ventanas deslizantes

1. Definición de la ventana deslizante:
    - Cada ventana tiene una duración fija de 5 minutos.
    - Las ventanas se deslizan en intervalos regulares (por ejemplo, cada minuto).
    - Esto significa que cada nueva ventana incluye parte de los datos de la ventana anterior y descarta los datos que ya no están dentro del rango de 5 minutos.
2. Procesamiento de los logs:
    - Los logs entran continuamente en el sistema como un flujo de datos.
    - Cada log tiene un timestamp asociado (por ejemplo, 2023-04-23T10:00:00).
    - El sistema evalúa si el timestamp del log cae dentro del rango de tiempo de la ventana actual.
3. Mantenimiento de los logs:

- Los logs que ya no están dentro del rango de los últimos 5 minutos se eliminan automáticamente.
- Por ejemplo, en la Ventana 1 (T10:00:00 a T10:04:00), el log con timestamp 2023-04-23T10:00:00 se elimina en la Ventana 2 porque ya no está dentro del rango de 5 minutos.

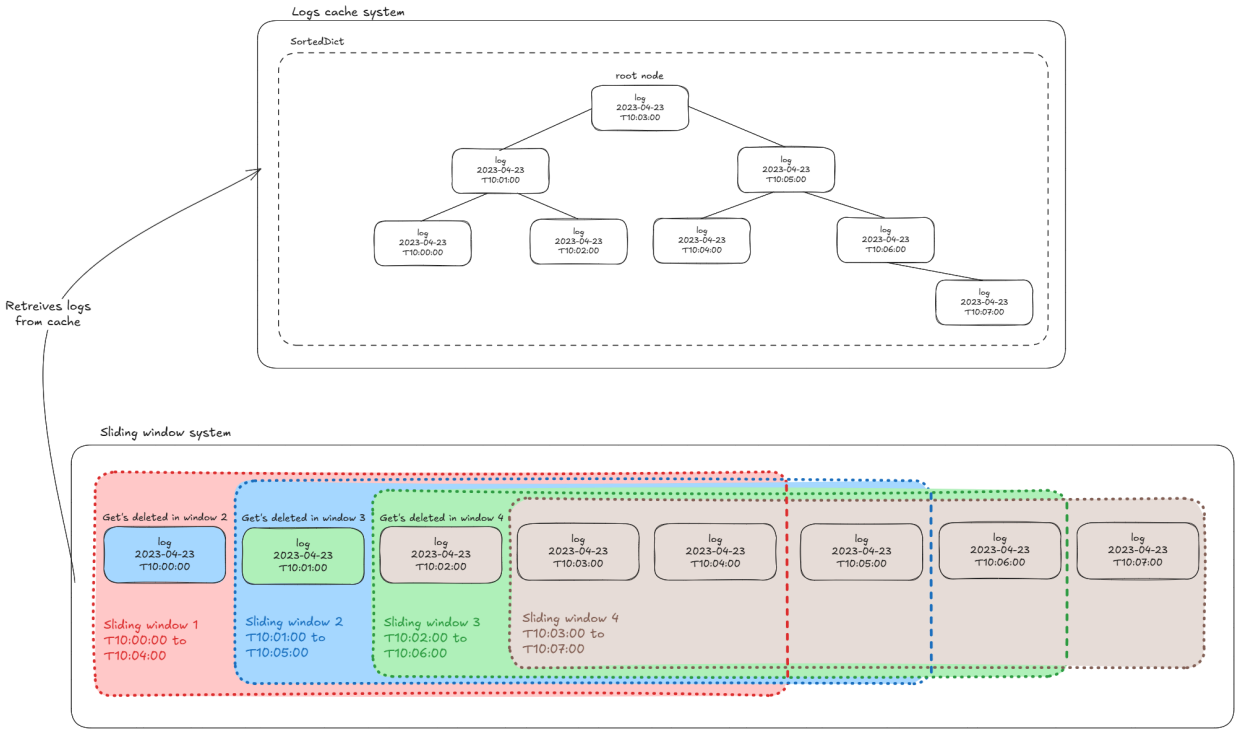
4. Superposición de ventanas:

- Las ventanas se superponen parcialmente, lo que permite que los logs recientes sean procesados en múltiples ventanas consecutivas.
- Por ejemplo, el log con timestamp 2023-04-23T10:03:00 aparece en la Ventana 3 y la Ventana 4.



Funcionamiento final

Finalmente, nuestro sistema de cache funcionara de la siguiente manera:

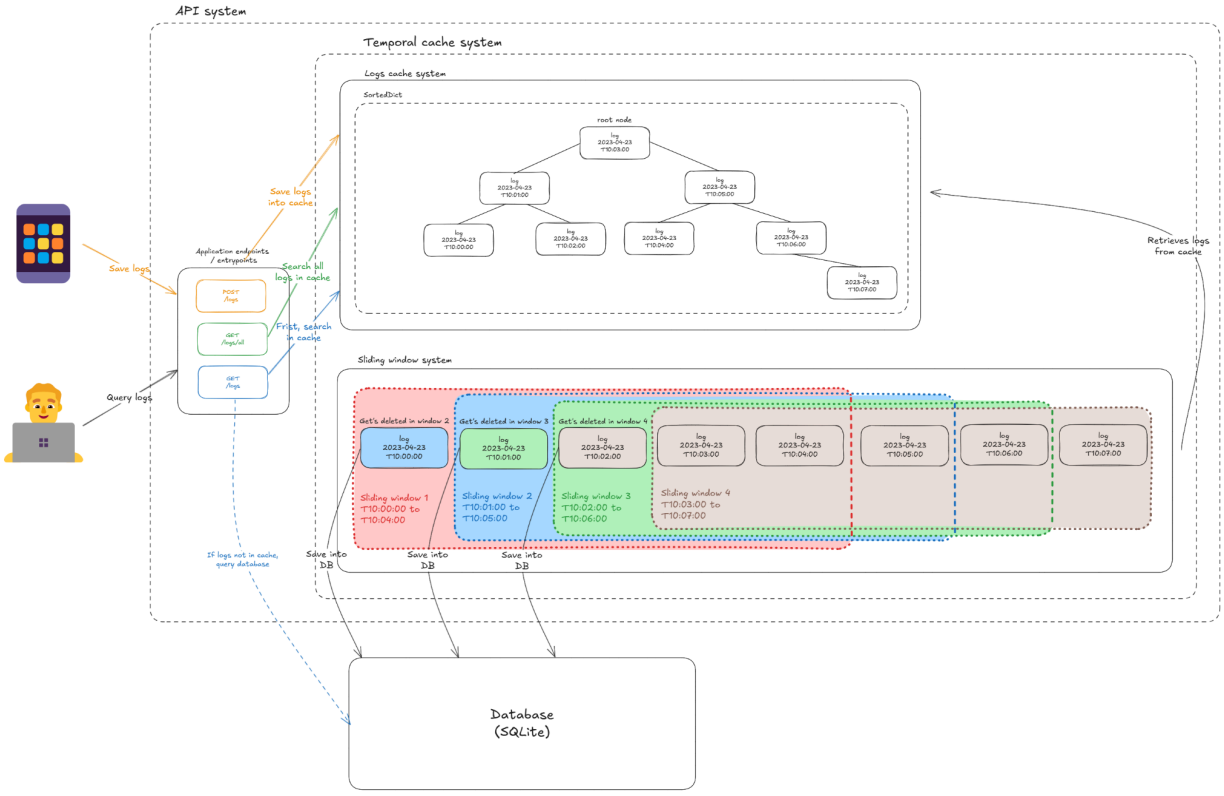


El sistema de almacenamiento de logs utiliza un `SortedDict` para almacenar los logs generados, donde cada nuevo log se agrega con su timestamp como clave. El `SortedDict` mantiene automáticamente los logs ordenados cronológicamente gracias a su estructura interna basada en un árbol balanceado, lo que asegura que los elementos siempre estén organizados. El sistema de ventana deslizante trabaja obteniendo los logs actuales del `SortedDict` dentro de un rango de tiempo específico. Para manejar estos logs de manera eficiente, se utiliza un `deque`, una estructura de datos que permite agregar, acceder y eliminar elementos rápidamente desde ambos extremos. Los logs

que caen dentro del rango de la ventana se agregan al deque, mientras que los que están fuera del rango se eliminan. Este enfoque asegura que el deque contenga solo los logs relevantes para el rango de tiempo actual, optimizando el procesamiento y el acceso a los logs en tiempo real. De esta manera, el sistema mantiene un caché de logs ordenado y eficiente, con la capacidad de ajustar dinámicamente el conjunto de logs que se procesan.

Arquitectura final

Finalmente, toda la arquitectura y funcionamiento de nuestro sistema se vería de esta forma:



¿Y la modularidad?

Como se puede observar, las responsabilidades del sistema —mencionadas anteriormente— se encuentran adecuadamente distribuidas en distintos paquetes y

módulos. Esta separación permite evitar la concentración de toda la lógica en un solo archivo, lo cual facilita la lectura, edición y mantenimiento del código, y previene que el crecimiento del sistema se convierta en un dolor de cabeza.

Usemos nuestro sistema