Informe de Diseño del Sistema Distribuido: Dispotify

Gabriel Herrera Carrazana y Adrian Alejandro Souto Morales , Diciembre 2025 .

Índice

Resumen

Dispotify es un sistema distribuido de streaming musical que implementa una arquitectura peer-to-peer coordinada mediante el algoritmo de consenso Raft. El sistema permite la gestión distribuida de archivos musicales con replicación automática, tolerancia a fallos y consistencia fuerte para operaciones críticas.

1. Arquitectura

La arquitectura es la base fundamental sobre la que se construye cualquier sistema distribuido robusto. Esta sección detalla la organización estructural, los roles funcionales y la estrategia de despliegue que constituyen el esqueleto del sistema.

1.1. Organización del Sistema Distribuido

La arquitectura de Dispotify es un modelo híbrido que fusiona la escalabilidad de un sistema peer-to-peer (P2P) con la consistencia fuerte de un modelo líder-seguidor, implementado a través del algoritmo de consenso Raft. Esta sinergia permite que todos los nodos operen como iguales, atendiendo solicitudes de clientes y comunicándose directamente entre sí para minimizar la latencia. Simultáneamente, para operaciones críticas como la modificación del catálogo musical, el protocolo Raft designa un líder temporal que ordena estas acciones, asegurando que se apliquen de manera consistente en todo el clúster.

1.2. Roles Existentes en el Sistema

El sistema se compone de dos capas principales claramente diferenciadas: el backend distribuido y el frontend de usuario.

El **backend** está formado por nodos multifuncionales que integran un conjunto completo de responsabilidades. Este enfoque de diseño, donde cada nodo tiene "capacidades idénticas", aumenta significativamente la resiliencia del sistema, ya que la pérdida de cualquier nodo individual no elimina una capacidad crítica del clúster. Cada instancia es un participante completo y autónomo. Los roles que cada nodo desempeña simultáneamente incluyen:

- **Servidor API**: Actúa como el punto de entrada para todas las solicitudes de clientes externos, exponiendo una API REST (FastAPI) para la gestión del catálogo musical.
- **Participante Raft**: Es un miembro activo del protocolo de consenso, capaz de operar como líder (coordinando operaciones) o como seguidor (replicando el estado del líder).
- **Gestor de Replicación**: Responsable de la distribución física de los archivos musicales a través del clúster, asegurando que se mantenga el factor de replicación configurado.
- **Cliente P2P Interno**: Facilita la comunicación entre los nodos, gestionando fallos de red transitorios mediante reintentos y timeouts.
- **Servicio de Descubrimiento**: Monitorea de forma continua la salud y disponibilidad de los otros nodos del clúster para mantener una visión actualizada de la topología.

Por su parte, el **frontend** es una aplicación web moderna desarrollada con React y TypeScript. Actúa como un cliente completamente externo que interactúa con el sistema exclusivamente a través de la API REST pública expuesta por los nodos del backend. Ofrece funcionalidades como la navegación del catálogo musical, reproducción de audio, búsqueda de canciones y carga de nuevos archivos musicales.

1.3. Distribución de Servicios en Docker

La estrategia de despliegue del sistema se basa en Docker Swarm, aprovechando su infraestructura de red para simplificar la comunicación distribuida. El despliegue consta de dos tipos de servicios:

- **Servicio Backend**: Múltiples réplicas de contenedores idénticos que forman el clúster distribuido. Cada contenedor ejecuta una instancia completa del servidor con todos los componentes de consenso, replicación y descubrimiento.
- **Servicio Frontend**: Uno o más contenedores que sirven la aplicación web React compilada.

El diseño se apoya de forma crítica en la red *overlay* de Swarm, que crea una capa de red virtual unificada para todos los contenedores. Esta red, junto a su sistema de DNS interno, actúa como el mecanismo fundamental para el descubrimiento y la comunicación transparente, permitiendo que

los nodos del backend se resuelvan entre sí mediante nombres de servicio como si estuvieran en la misma máquina.

La arquitectura física y lógica del sistema cobra vida a través de los procesos concurrentes que se ejecutan dentro de cada nodo, los cuales se detallan a continuación.

2. Procesos

La gestión de procesos y la concurrencia son vitales para el rendimiento y la eficiencia de un sistema de alto rendimiento como Dispotify. Un manejo inadecuado de la concurrencia puede conducir a cuellos de botella y a un uso ineficiente de los recursos, comprometiendo la capacidad de respuesta del sistema bajo carga.

2.1. Tipos de Procesos dentro del Sistema

El modelo de concurrencia del sistema se basa en un único proceso principal (el servidor FastAPI) que orquesta múltiples corrutinas asíncronas de Python. Estos componentes lógicos operan como "procesos" cooperativos dentro del mismo espacio de memoria, lo que permite una comunicación interna de muy baja latencia y un uso eficiente de la CPU, ya que el tiempo de procesador se cede cooperativamente solo durante operaciones de entrada/salida (I/O).

Los componentes lógicos clave que operan como procesos cooperativos son:

- **Servidor API**: Gestiona el ciclo de vida de las solicitudes HTTP entrantes.
- **Máquina de Estados Raft**: Procesa los eventos del protocolo de consenso y mantiene el estado distribuido.
- **Gestor de Replicación de Archivos**: Orquesta la distribución y verificación de réplicas de archivos en segundo plano.
- **Cola de Eventos Interna**: Facilita la comunicación desacoplada y asíncrona entre los diferentes componentes.

2.2. Organización o Agrupación de los Procesos

Para Dispotify, hemos adoptado un patrón de diseño que denominamos "monolítico inteligente". A pesar de que todos los componentes funcionales se ejecutan dentro de un único contenedor Docker, mantienen una separación lógica de responsabilidades bien definida. Esta decisión estratégica prioriza la simplicidad de despliegue y la eficiencia de la comunicación interna sobre la aislación física de los microservicios, lo que consideramos un trade-off beneficioso para este caso de uso.

Este enfoque ofrece ventajas significativas:

- **Simplicidad de Despliegue**: Un único artefacto de despliegue simplifica la gestión operativa, la monitorización y el ciclo de vida del servicio.
- **Eficiencia de Comunicación Interna**: Al operar en un mismo espacio de memoria, la comunicación es extremadamente eficiente, utilizando estado compartido y llamadas a funciones directas en lugar de mecanismos más lentos como la serialización de red.
- **Inicialización Coordinada**: El proceso principal es responsable de iniciar todos los componentes internos en la secuencia correcta, garantizando un arranque ordenado y predecible del nodo.

- **Tareas Asíncronas en Segundo Plano**: Operaciones de larga duración, como el monitoreo de salud o la limpieza de datos, se ejecutan como corrutinas en segundo plano, sin interferir con el procesamiento de solicitudes de la API.

2.3. Tipo de Patrón de Diseño con Respecto al Desempeño

Para lograr un alto rendimiento bajo cargas concurrentes, hemos implementado el patrón de diseño **Reactor**. Este patrón se materializa a través del *event loop* de `asyncio` de Python, que actúa como un despachador de eventos central. Permite que un único hilo de ejecución gestione miles de conexiones y tareas concurrentes de manera eficiente mediante el uso de operaciones de I/O no bloqueantes.

El impacto de este patrón en el rendimiento se resume en los siguientes puntos:

- **Multitarea Cooperativa**: Un único *event loop* orquesta la ejecución de múltiples corrutinas, maximizando el uso de la CPU al eliminar los tiempos de espera ociosos.
- **I/O No Bloqueante**: Las operaciones de red y disco no detienen el procesamiento de otras solicitudes, lo que es crucial para un sistema de streaming.
- **Gestión de Conexiones**: Se utiliza *connection pooling* para reutilizar conexiones TCP, reduciendo la latencia y la sobrecarga asociadas a su creación.
- **Tolerancia a la Latencia**: Todas las operaciones de red implementan *timeouts* configurables para evitar que un nodo lento degrade el rendimiento de todo el sistema.

Esta organización de procesos depende de mecanismos de comunicación eficientes para su interacción, estableciendo el vínculo con la siguiente sección.

3. Comunicación

La comunicación es el sistema nervioso de cualquier sistema distribuido; su eficiencia y robustez determinan la capacidad de respuesta y la fiabilidad del conjunto. Dispotify adopta una arquitectura de comunicación dual (HTTP/REST y HTTP RPC) como decisión estratégica para optimizar diferentes tipos de interacción, utilizando el protocolo más adecuado para cada caso de uso.

3.1. Tipo de Comunicación

El sistema combina protocolos y formatos para equilibrar la interoperabilidad, la legibilidad y el rendimiento.

Protocolo/ Formato	Caso de Uso Principal
HTTP/REST	API pública para clientes externos.
HTTP/RPC	Comunicación interna entre nodos.

3.2. Comunicación Cliente-Servidor y Servidor-Servidor

El sistema distingue claramente entre las interacciones con clientes externos y la comunicación interna del clúster.

Comunicación Cliente-Servidor: Se realiza exclusivamente a través de la API pública HTTP/REST. El frontend de React consume esta API para todas sus operaciones, siguiendo el patrón tradicional de solicitud-respuesta.

- `/music/upload`: Endpoint para que los usuarios carguen nuevos archivos musicales.

Comunicación Servidor-Servidor: Utiliza HTTP para optimizar el rendimiento de las operaciones internas críticas del clúster. El cliente HTTP interno implementa lógicas avanzadas de reintentos con *backoff* exponencial y manejo de fallos transitorios.

- `ReplicateFile`: Servicio HTTP para instruir a otros nodos a replicar un archivo mediante streaming eficiente.
- `RaftService`: Conjunto de métodos HTTP dedicados a la comunicación del protocolo de consenso Raft (`RequestVote`, `AppendEntries`, etc.).

3.3. Comunicación entre Procesos

Gracias al diseño de contenedor único, la comunicación entre los componentes lógicos dentro de un mismo nodo es altamente eficiente, evitando la sobrecarga de los protocolos de red.

- **Estado Compartido**: La máquina de estados Raft centraliza el estado crítico del clúster, permitiendo un acceso rápido y consistente por parte de otros módulos.
- **Callbacks**: Se utilizan para notificaciones síncronas donde un componente necesita reaccionar inmediatamente a un evento, como un cambio en el liderazgo.

Estos protocolos de comunicación permiten que los nodos interactúen, pero es la capa de coordinación la que asegura que estas interacciones resulten en un estado global consistente.

4. Coordinación

La coordinación es el desafío central en el diseño de sistemas distribuidos, ya que es el mecanismo que garantiza que un conjunto de nodos independientes pueda actuar como un sistema único y coherente. Dispotify deposita su confianza en el algoritmo de consenso Raft como la piedra angular para lograr coherencia y tolerancia a fallos.

4.1. Sincronización de Acciones

El algoritmo Raft garantiza un orden total y secuencial de las operaciones de escritura al designar a un único nodo, el **líder**, como el único autorizado para iniciarlas. Cualquier operación que modifique el estado se registra en un log distribuido y solo se confirma cuando una mayoría de nodos ha replicado la entrada, garantizando que el cambio sea duradero y consistente.

Los mecanismos específicos que facilitan esta sincronización son:

- **Elección de Líder**: Un proceso automático y tolerante a fallos que utiliza timeouts aleatorios para elegir un nuevo líder si el actual falla.
- **Heartbeats Periódicos**: El líder envía mensajes periódicos a los seguidores para mantener su autoridad y permitir que el clúster detecte rápidamente su fallo.
- **Sincronización Automática**: Los nodos que se desconectan temporalmente o se unen al clúster se ponen al día automáticamente replicando el log del líder.

4.2. Acceso Exclusivo a Recursos. Condiciones de Carrera

Para prevenir condiciones de carrera, Dispotify implementa una estrategia de **Coordinación por Líder Raft** :

- **Coordinación Centralizada en el Líder**: Solo el nodo líder puede iniciar y coordinar operaciones críticas de mantenimiento o modificación de recursos compartidos.
- **Asignación de Trabajo Dirigida**: El líder asigna tareas específicas a nodos específicos de manera determinista y ordenada.
- **Registro en el Log de Raft**: Cada asignación o autorización se registra como una entrada en el log distribuido de Raft, que actúa como la fuente de verdad única y consistente para el estado de las operaciones en curso.
- **Timeouts de Operaciones Activas**: Cada operación activa registrada incluye un timeout configurable. Si la operación no se completa en el tiempo esperado, expira automáticamente, previniendo bloqueos indefinidos (*deadlocks*) y permitiendo que el líder pueda reasignar la tarea si es necesario.
- **Verificación de Titularidad y Estado**: Solo el nodo al que se le asignó una tarea puede reportar su finalización o modificar su estado, previniendo interferencias no autorizadas.

4.3. Toma de Decisiones Distribuidas

El modelo de toma de decisiones se basa en el consenso por **mayoría simple** de Raft. Cualquier cambio de estado global debe ser propuesto por el líder y replicado en una mayoría (`N/2 + 1`) de los nodos antes de ser considerado válido.

Además, el sistema utiliza algoritmos deterministas para tomar decisiones distribuidas:

- **Consistent Hashing con Nodos Virtuales**: Se utiliza para decidir dónde deben ubicarse las réplicas de los archivos. El sistema aplica el algoritmo **MD5** al `file_id` del archivo para mapearlo a una posición en un anillo de hash. El uso de nodos virtuales por cada nodo físico mejora la distribución uniforme de los datos.
- **Términos Raft**: Actúan como un reloj lógico, ayudando a ordenar eventos y resolver conflictos entre operaciones propuestas en diferentes momentos.

Una vez coordinadas las acciones, es crucial poder nombrar y localizar los recursos sobre los que se actúa.

5. Nombrado y Localización

En un sistema distribuido, donde los recursos están físicamente dispersos, un esquema de nombrado y localización claro y robusto es fundamental. Dispotify emplea una combinación de identificadores únicos, almacenamiento híbrido y descubrimiento dinámico para asegurar que todos los recursos puedan ser identificados y accedidos de manera eficiente.

5.1. Identificación de los Datos y Servicios

El sistema utiliza esquemas de identificación bien definidos para garantizar la unicidad y la integridad. Es crucial distinguir el propósito de los diferentes algoritmos de hash:

- **SHA256**: Se utiliza para la **verificación de integridad del contenido**. Es un hash criptográficamente seguro que garantiza que los archivos no han sido alterados.
- **MD5**: Se utiliza exclusivamente para la **distribución de archivos** en el anillo de *Consistent Hashing*. No se usa con fines de seguridad.

Los identificadores clave del sistema son:

- **Archivos**: Identificados por un `file_id` único, derivado del nombre del archivo. Su integridad se verifica con un checksum **SHA256** de su contenido, y su ubicación se determina aplicando **MD5** al `file_id`.
- **Nodos**: Cada nodo tiene un `node_id` único y persistente.
- **Servicios**: Los endpoints de la API están organizados en namespaces lógicos (ej. `/music/`, `/internal/`) que separan las funcionalidades.

5.2. Ubicación de los Datos y Servicios

La estrategia de almacenamiento es híbrida, diseñada para optimizar el acceso según el tipo de dato.

Tipo de Dato	Ubicación de Almacenamiento
Metadatos	Instancias PostgreSQL locales en cada nodo.
Archivos Binarios	Sistema de archivos distribuido entre los nodos.
Estado de Consenso Raft	Logs persistentes en el sistema de archivos local.

5.3. Localización de los Datos y Servicios

Para garantizar una alta disponibilidad, hemos diseñado un sistema de descubrimiento híbrido y de múltiples capas, que es mucho más robusto que depender únicamente de DNS.

1. **Bootstrap con DNS de Docker**: Al arrancar, un nodo utiliza el DNS interno de Docker para una resolución inicial de los nombres de servicio de sus pares. Este es solo el primer paso.
2. **Registro en el Líder Raft**: Inmediatamente después, el nuevo nodo se registra explícitamente con el líder del clúster Raft. Esta información de membresía se propaga a todos los nodos a través del log de consenso, convirtiendo a Raft en la fuente de verdad autoritativa de la topología del clúster.
3. **Descubrimiento Activo y Caché Local**: Cada nodo ejecuta un servicio de descubrimiento activo que realiza *health checks* HTTP periódicos (cada 5-10 segundos) a los demás miembros. Esto mantiene una caché local de direcciones IP de pares con un TTL corto, garantizando una visión casi en tiempo real del estado de salud del clúster.
4. **Fallback Robusto**: El cliente de comunicación interno está diseñado para ser resiliente, siguiendo una secuencia de fallback: intenta resolver usando el DNS, si falla, consulta su tabla de routing local (la caché actualizada por los health checks) y, si es necesario, aplica reintentos.

Finalmente, la localización de un archivo específico es **determinista** gracias al *Consistent Hashing*, eliminando la necesidad de realizar búsquedas globales. Simplemente aplicamos MD5 sobre el `file_id` del archivo para determinar qué nodos deben almacenar sus réplicas.

La localización de los datos está intrínsecamente ligada a las estrategias de consistencia y replicación que los gobiernan.

6. Consistencia y Replicación

La consistencia y la replicación son las garantías gemelas que aseguran la durabilidad y la corrección de los datos. La replicación proporciona redundancia para sobrevivir a fallos, mientras que la consistencia asegura que todas las réplicas presenten una visión coherente. Dispotify adopta un enfoque pragmático, aplicando diferentes modelos según la criticidad de cada tipo de dato.

6.1. Modelos de Consistencia Aplicados

Adoptamos un enfoque "políglota" para la consistencia, aplicando el modelo más adecuado para cada tipo de dato:

- **Consistencia Fuerte**: Se aplica al **estado del clúster** (membresía, topología) y a los **metadatos de las canciones** (almacenados en PostgreSQL). Todas las operaciones de escritura deben ser confirmadas por una mayoría del clúster Raft antes de ser aplicadas, garantizando que cualquier lectura posterior refleje el último estado confirmado.
- **Consistencia Híbrida para Archivos Binarios**:
    - **Escritura**: Se rige por un modelo de consistencia fuerte. Una operación de escritura solo se considera exitosa tras recibir confirmación de un quórum de réplicas.
    - **Lectura**: Sigue un modelo de consistencia eventual. Las lecturas pueden servirse desde cualquier réplica disponible, con mecanismos de versionado para detectar y actualizar réplicas desactualizadas en segundo plano.

6.2. Estrategia de Replicación

El mecanismo de replicación de archivos está diseñado para ser robusto y flexible.

- **Factor de Replicación (R)**: El número de copias de cada archivo es configurable, con un valor por defecto de **R=3**. Este valor ofrece un buen equilibrio entre durabilidad y coste de almacenamiento.
- **Iniciación por el Líder**: Para evitar conflictos, el proceso de replicación de un nuevo archivo solo puede ser iniciado por el líder del clúster Raft, que centraliza la decisión y asegura un orden consistente.
- **Confirmación por Quórum de Réplicas**: Una operación de escritura de un archivo requiere la confirmación de un quórum de `(R+1)/2` réplicas. Para R=3, se necesitan 2 confirmaciones. Es importante destacar que este quórum de réplicas es independiente del quórum del clúster Raft.

6.3. Fiabilidad de las Réplicas de los Datos

El sistema implementa múltiples capas de verificación para garantizar que las réplicas no solo existan, sino que sean correctas.

- **Verificación por Checksums SHA256**: Asegura la integridad bit a bit de los archivos durante la transferencia y en reposo. El checksum se compara antes y después de cada transferencia para detectar cualquier corrupción.
- **Recuperación Automática**: Si se detecta una réplica corrupta, el sistema la marca como inválida e inicia automáticamente un proceso de recuperación, replicando el archivo desde una copia válida conocida.
- **Resincronización Post-Fallo**: Los nodos que se recuperan de un fallo verifican la integridad de sus datos locales y los sincronizan con el estado actual del clúster antes de volver a operar plenamente.

Estas estrategias de replicación son la base de la capacidad del sistema para tolerar fallos.

7. Tolerancia a Fallos

En los sistemas distribuidos, los fallos no son una eventualidad, sino una certeza. Por ello, la tolerancia a fallos es un requisito de diseño fundamental. Dispotify está diseñado con la expectativa de que los nodos y la red fallarán, tratando los errores como parte de su operación normal.

7.1. Respuesta a Errores

El sistema sigue una filosofía de "fallo elegante", con estrategias proactivas para diferentes tipos de errores. Es crucial entender que el sistema está diseñado para manejar fallos a dos niveles distintos: el de consenso, que afecta la capacidad del clúster para tomar decisiones, y el de datos, que afecta la disponibilidad de archivos específicos.

- **Manejo de Fallos Transitorios**: Para problemas como una pérdida momentánea de conectividad, el cliente de comunicación interno utiliza reintentos con *backoff* exponencial, evitando saturar un nodo que podría estar recuperándose.
- **Manejo de Fallos Permanentes**: Ante la caída de un servidor, el protocolo Raft se encarga de elegir automáticamente un nuevo líder, y el servicio de descubrimiento actualiza la topología del clúster para redirigir el tráfico.

En caso de **particiones de red**, el protocolo Raft asegura que solo la partición que contenga a la mayoría de los nodos (`> N/2`) siga operativa y pueda procesar escrituras. La partición minoritaria queda en estado de espera. Al resolverse la partición, los nodos de la partición minoritaria se resincronizan automáticamente replicando las entradas del log que acumularon como retraso. No se producen logs divergentes, ya que la partición minoritaria no pudo confirmar nuevas escrituras.

7.2. Nivel de Tolerancia a Fallos

El nivel de tolerancia a fallos del sistema es cuantificable y se diferencia entre el clúster de consenso y la disponibilidad de los datos.

- **Tolerancia del Clúster Raft**: Con `N` nodos, el sistema puede tolerar hasta **`f = ⌊(N-1)/2⌋`** fallos simultáneos sin perder la capacidad de alcanzar consenso para operaciones de escritura.
- **Disponibilidad de Datos**: Depende del factor de replicación **`R`** (por defecto 3), que determina cuántas copias de cada archivo existen. Un archivo permanece disponible para lectura mientras al menos una de sus réplicas esté en línea.

7.3. Gestión de Cambios Dinámicos en el Clúster

El sistema maneja con fluidez los cambios en la membresía del clúster.

- **Sincronización Automática**: Cuando un nodo se reincorpora tras una caída temporal, inicia un proceso de resincronización en el que se pone al día con todos los cambios que ocurrieron durante su ausencia.
- **Bootstrap Progresivo**: La incorporación de nodos completamente nuevos se realiza de forma gradual. En lugar de una transferencia masiva de datos, la carga de replicación se asigna al nuevo nodo de forma incremental para minimizar el impacto en el rendimiento del clúster.

La robustez ante fallos debe complementarse con una sólida estrategia de seguridad para proteger el sistema de amenazas externas.

8. Seguridad

La seguridad es una dimensión crítica que abarca la protección de datos y el control de acceso. Este informe aborda el estado de la seguridad de manera transparente, distinguiendo entre las características inherentes al diseño y las áreas identificadas para futuras mejoras.

8.1. Seguridad con Respecto a la Comunicación

Actualmente, la comunicación interna confía en el aislamiento proporcionado por la red privada de Docker Swarm, mientras que el cifrado externo (TLS/HTTPS) se delega a un componente externo como un proxy inverso.

El plan de desarrollo en este ámbito incluye las siguientes iniciativas activas:

- **TLS para comunicación interna**: Actualmente en proceso de implementación para cifrar todo el tráfico entre nodos.
- **Autenticación Mutua entre Nodos**: Planificada para permitir que los nodos verifiquen criptográficamente la identidad de los demás.
- **Cifrado de Datos en Reposo**: Actualmente en proceso de implementación para proteger los archivos musicales almacenados en disco.

8.2. Seguridad con Respecto al Diseño

La propia arquitectura de Dispotify proporciona varias fortalezas de seguridad inherentes:

- **Arquitectura Descentralizada**: La ausencia de un punto único de fallo (SPOF) mejora tanto la disponibilidad como la seguridad, ya que comprometer un solo nodo no compromete la totalidad del sistema.
- **Transferencia Directa P2P**: Durante la replicación, los archivos se transfieren directamente entre nodos, lo que reduce la superficie de ataque.
- **Consenso por Mayoría**: El protocolo Raft protege contra cambios de estado maliciosos iniciados por una minoría de nodos comprometidos.
- **Verificación de Integridad de Datos**: El uso de checksums **SHA256** protege los archivos contra la corrupción, ya sea accidental o malintencionada. Es importante reiterar que **SHA256** se usa para seguridad e integridad, mientras que **MD5** se reserva exclusivamente para el algoritmo de distribución no seguro de *Consistent Hashing*.

8.3. Autorización y Autenticación

El enfoque actual del control de acceso se basa principalmente en el aislamiento a nivel de red.

Capacidades Actuales	Hoja de Ruta de Desarrollo
Aislamiento de red de Docker Swarm.	Sistema de autenticación de usuarios.
Control de acceso por firewall.	Control de acceso basado en roles (RBAC).
	Logs de auditoría de operaciones.
	Soporte para tokens de API.