ADR 001: Uso de Sesión HTTP Compartida (`ClientSession`)

- **Estado:** Aceptado
- **Fecha:** 2023-10-27

Contexto

El dashboard requiere realizar múltiples peticiones HTTP a diversos endpoints para componer la interfaz. La creación de una conexión TCP/TLS es costosa en términos de tiempo (handshake) y recursos de CPU. Se requiere una estrategia para gestionar estas conexiones de manera eficiente en un entorno de alta concurrencia.

Decisión

Utilizar una única instancia de `aiohttp.ClientSession` compartida para todo el ciclo de vida de la aplicación del dashboard.

Alternativas

- **Sesión por petición:** Crear y cerrar una `ClientSession` para cada llamada individual.
- **Pools aislados:** Una sesión para APIs internas críticas y otra sesión para servicios externos de terceros.

Consecuencias

- **Positivas:** Maximiza el *Connection Pooling* (reutilización de conexiones keep-alive), reduciendo drásticamente la latencia de red y la carga en el servidor.
- **Negativas:** Introduce un punto único de fallo. Si un servicio externo lento satura el pool de conexiones de la sesión, puede bloquear peticiones a servicios internos rápidos (Head-of-Line Blocking a nivel de pool). Existe riesgo de compartir estado mutable (cookies) incorrectamente si no se gestiona con cuidado.

ADR 002: Orquestación de Peticiones con `asyncio.gather`

- **Estado:** Aceptado (Con reservas de UX identificadas)
- **Fecha:** 2023-10-27

Contexto

El dashboard debe cargar datos de múltiples fuentes independientes simultáneamente. Se necesita un mecanismo para esperar la finalización de estas tareas asíncronas antes de presentar el estado final o manejar errores.

Decisión

Utilizar `asyncio.gather(..., return_exceptions=True)` para agrupar las corrutinas.

Alternativas

- **`asyncio.as_completed()`:** Procesar y renderizar cada resultado tan pronto como esté disponible.
- **`asyncio.wait()`:** Esperar por condiciones específicas (FIRST_COMPLETED, ALL_COMPLETED).
- **Ejecución serial:** Ejecutar una tras otra (descartado por rendimiento).

Consecuencias

- **Positivas:** Simplifica el manejo de errores al devolver una lista unificada de resultados/excepciones. Código más limpio y fácil de leer para procesamiento "batch".
- **Negativas: Latencia Percibida:** La interfaz se bloquea hasta que la *más lenta* de las peticiones finaliza. No permite "carga progresiva", lo cual degrada la experiencia de usuario (UX) si una sola petición tiene un *outlier* de latencia.

ADR 003: Estrategia de Timeouts Individuales

- **Estado:** Aceptado
- **Fecha:** 2023-10-27

Contexto

Las peticiones de red pueden quedarse "colgadas" indefinidamente debido a problemas de servidor o red, consumiendo recursos y manteniendo al usuario en espera.

Decisión

Configurar un `ClientTimeout` específico para cada petición individual (e.g., 5s, 10s), sin un timeout global que envuelva el grupo de tareas.

Alternativas

- **Timeout Global:** Envolver el `asyncio.gather` en un `wait_for` global.
- **Timeout por defecto de aiohttp:** Usar los valores por defecto (generalmente 5 min, demasiado largo).

Consecuencias

- **Positivas:** Permite granularidad; una petición pesada (reporte) puede tener más tiempo que una ligera (ping). Evita que una tarea lenta cancele tareas rápidas que ya tuvieron éxito.
- **Negativas:** Si se usa con `gather`, el usuario podría esperar la suma del timeout más largo configurado, lo cual puede exceder su paciencia (e.g., esperar 10s cuando el usuario abandona a los 3s). No garantiza un tiempo máximo de respuesta para la vista completa.

ADR 004: Límite de Concurrencia Estático (Semáforo)

- **Estado:** Aceptado
- **Fecha:** 2023-10-27

Contexto

El cliente no debe saturar los recursos locales (descriptores de archivo) ni realizar un ataque de denegación de servicio involuntario (DoS) contra el servidor al lanzar demasiadas peticiones simultáneas.

Decisión

Implementar un `asyncio.Semaphore(10)` que envuelva las llamadas a la API.

Alternativas

- **Sin límite:** Lanzar todo lo que el loop de eventos permita.
- **Límite Dinámico (Backpressure):** Ajustar el límite basándose en la tasa de errores o latencia del servidor.

Consecuencias

- **Positivas:** Protege al cliente de `OOM` (Out of Memory) y al servidor de picos repentinos de tráfico. Fácil de implementar.
- **Negativas:** El número 10 es arbitrario ("número mágico"). Puede ser un cuello de botella artificial si el servidor y la red pueden manejar más carga (sub-utilización), o insuficiente si el servidor ya está degradado. No se adapta a las condiciones cambiantes del entorno.

ADR 005: Política de No-Reintentos (Fail-Fast)

- **Estado:** Aceptado
- **Fecha:** 2023-10-27

Contexto

Los fallos de red son inevitables. Se debe decidir si el cliente debe intentar recuperarse automáticamente de errores transitorios.

Decisión

No implementar lógica de reintentos automáticos (retries) en la primera versión. Si una petición falla, se muestra el error inmediatamente.

Alternativas

- **Reintentos inmediatos:** Reintentar sin espera (peligroso).
- **Exponential Backoff:** Reintentar con espera incremental.

Consecuencias

- **Positivas: Predictibilidad de Latencia:** Evita que el usuario espere tiempos excesivos debido a múltiples intentos fallidos (especialmente crítico al usar `gather`). Evita el problema de "Thundering Herd" (amplificación de tráfico) sobre un servidor ya saturado.
- **Negativas:** Menor resiliencia ante micro-cortes de red ("blips"). El usuario verá errores por fallos que podrían haberse resuelto con un simple reintento de 100ms.