ADR-001: Arquitectura y Refactorización del Cliente HTTP

- **Estado:** Aceptado / Propuesto para Refactorización
- **Fecha:** 05-Feb-2026
- **Contexto:** El proyecto requiere un cliente HTTP para comunicarse con la API de EcoMarket. La implementación inicial priorizó la simplicidad (MVP) utilizando funciones independientes, validación manual y configuraciones rígidas. Se ha identificado la necesidad de mejorar la eficiencia, la resiliencia ante fallos de red y la mantenibilidad del código a medida que la API evoluciona.

Decisión 1: Persistencia de Conexiones (Session vs. Peticiones Aisladas)

Contexto

Actualmente, cada llamada a función abre una nueva conexión TCP/TLS (`requests.get`), lo que añade latencia significativa por el *handshake* en cada operación.

Decisión

Implementar un objeto `requests.Session` compartido.

Aunque se mantendrá la estructura de funciones independientes (evitando clases complejas), estas funciones operarán sobre una instancia de `Session` reutilizable (pasada como argumento o instanciada a nivel de módulo/paquete).

Alternativas

- **Estado actual (Nueva conexión por llamada):** Rechazada por ineficiencia y latencia alta.
- **Clase Singleton:** Rechazada para mantener la simplicidad funcional y evitar el "boilerplate" de POO innecesario en Python.

Consecuencias

- **Positivas:** Reducción drástica de latencia gracias a *TCP Keep-Alive* (reutilización de conexiones). Gestión centralizada de cookies y headers comunes.
- **Negativas:** Introduce un estado mutable (la sesión) que debe ser gestionado correctamente (ej. cerrarse al finalizar la aplicación).

Decisión 2: Estrategia de Validación de Datos

Contexto

La validación actual depende de bloques imperativos `if/else` manuales para verificar la existencia y tipos de claves en el JSON de respuesta. Esto es propenso a errores y difícil de escalar.

Decisión

Migrar a Pydantic para validación declarativa.

Se definirán modelos (schemas) que representen la estructura esperada de la respuesta. El cliente parseará el JSON directamente contra estos modelos.

Alternativas

- **Validación Manual (`if/else`):** Rechazada por deuda técnica alta y dificultad de lectura.
- **`dataclasses` nativas:** Considerada, pero Pydantic ofrece mejor coerción de tipos y validación de datos externos.

Consecuencias

- **Positivas:** Garantía de tipos en tiempo de ejecución, código más limpio, autocompletado en el IDE y mensajes de error de validación detallados automáticos.
- **Negativas:** Añade una dependencia externa (`pydantic`) al proyecto. Curva de aprendizaje inicial mínima.

Decisión 3: Resiliencia (Reintentos y Timeouts)

Contexto

El cliente actual falla inmediatamente ante errores transitorios y tiene un timeout fijo de 10s, lo que lo hace frágil ante inestabilidad de red y rígido ante endpoints lentos.

Decisión

1. **Reintentos Acotados:** Configurar `HTTPAdapter` en la sesión con una estrategia de *Exponential Backoff* (máximo 3 reintentos) para errores de conexión y códigos 5xx.
2. **Timeouts Configurables:** Establecer 10s como valor por defecto, pero permitir sobrescribir este valor mediante un argumento `timeout` en cada función.

Alternativas

- **Sin reintentos:** Rechazada por fragilidad operativa.
- **Reintentos infinitos:** Rechazada por riesgo de causar Denegación de Servicio (DoS) al servidor.
- **Timeout Fijo:** Rechazada por falta de flexibilidad para operaciones pesadas futuras (ej. reportes).

Consecuencias

- **Positivas:** El sistema se auto-recupera de micro-cortes de red sin intervención humana. Se evita saturar al servidor gracias al límite de intentos. Flexibilidad para soportar operaciones lentas.
- **Negativas:** El tiempo total de respuesta ante un fallo real aumenta (suma de los tiempos de reintento).

Decisión 4: Manejo de Errores (Excepciones)

Contexto

Es necesario que el cliente reporte errores en términos del dominio de negocio, no solo errores HTTP genéricos, sin perder la trazabilidad técnica para depuración.

Decisión

Mantener el uso de Excepciones Personalizadas con Encadenamiento (`raise from e`).

Se capturan excepciones de la librería (ej. `requests.ConnectionError`) y se relanzan como excepciones de dominio (ej. `EcoMarketConnectionError`), preservando el *stack trace* original.

Alternativas

- **Exponer excepciones de `requests`:** Rechazada porque acopla el código del usuario a la librería de transporte.
- **Excepciones personalizadas sin `from e`:** Rechazada porque oculta la causa raíz del error, dificultando el *debugging*.

Consecuencias

- **Positivas:** Clara separación de responsabilidades. El consumidor del cliente puede manejar errores de negocio semánticos. Facilidad para cambiar la librería HTTP subyacente en el futuro sin romper el código cliente.
- **Negativas:** Requiere mantener una jerarquía de clases de excepción adicional.