Handler de autorización (documento uso interno)

Solicitantes: Carlos Isaac, Domingo Cordero, Juan Pasteris, Diego Soler, Carlos Torres

Objetivo

- Describir en forma resumida el problema que se intenta resolver
- Describir los componentes de la solución
- Describir las decisiones de diseño funcional y técnica

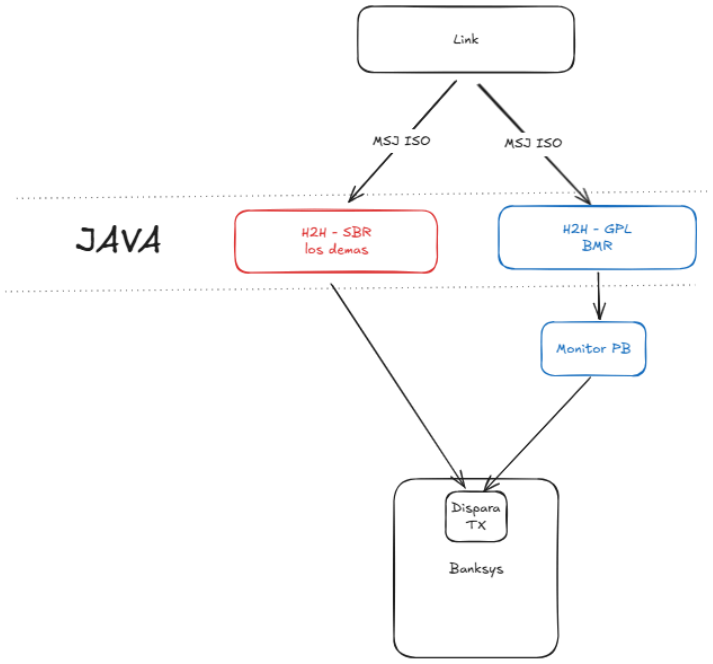
Problema que se intenta resolver

Actualmente, cuando se recibe desde LINK un mensaje de ISO, el mismo se descompone, parsea y se transforma en una aplicación JAVA conocida como MONITOR H2H. Logrando por este proceso la información necesaria, para el disparo de un proceso de la base de datos conocido por su nombre: sgl_dispara_transac_h2h. Este proceso público de la base de datos, tiene toda la lógica necesaria, para procesar la solicitud proveniente desde LINK.

El tiempo de respuesta de esta pila de llamadas desde la llegada del mensaje al MONITOR H2H, hasta la respuesta de la base de datos es crítico.

De no respetarse un tiempo, LINK al tener su propio saldo actualizado desde un proceso de sincronización diario llamado REFRESH, procede a autorizar por sí mismo, enviando una solicitud nueva al monitor conocido como transacción forzada.

Este mecanismo resuelve el problema de cara al cliente que está solicitando el movimiento, pero acarrea tráfico adicional de mensajería de reversas y transacciones forzadas, para ajustar los desacoples.



Este problema también se acelera por el flujo de llegada de solicitudes desde LINK. Este último sólo remite las solicitudes sin importar el caudal. Al no recibir respuesta en el tiempo esperado, procede a responder con su saldo, cancelar las solicitudes enviadas y enviarlas como forzadas.

Componentes de la solución

Para dar una solución a este problema se pensó en un conjunto de componentes que actuando juntos, permitan incorporar un componente que reciba las solicitudes y responda dentro de los parámetros requeridos, actualizando su saldo en un repositorio propio de forma similar a lo que hace el propio LINK, para luego gestionar la solicitud con el CORE.

La idea es que se intercepten las solicitudes que salen del MONITOR H2H, y se analice el saldo de la cuenta, si la misma tiene saldo suficiente, se acepta el movimiento registrandolo en un repositorio, en una cola, y actualizando el saldo. Luego en un momento posterior, se procesa en forma asincrónica el mensaje de la cola contra el CORE.

Esta solución implica al menos 3 componentes:

- **Handler de autorización**: Su misión es interceptar los mensajes provenientes del MONITOR H2H, analizar su saldo, si es suficiente acepta el movimiento, genera el registro en repositorio y agrega a las colas de eventos el mensaje de solicitud para su procesamiento. Si no es suficiente, rechaza el movimiento por falta de saldo, devolviendo esto dentro del margen de tiempo especificado por LINK.
- **Worker de solicitud**: Su misión es recuperar eventos de la cola asignada, disparar el proceso del CORE, analizar su resultado y actualizar estado (solicitud) y el saldo (cuenta) del handler.
- **Worker de sincronización**: Su misión es la de mantener el saldo de handler actualizado con el saldo del CORE. Por modificaciones internas en CORE (otros canales) pueden llegar alteraciones del saldo. Como el saldo del CORE es prioritario, entonces este saldo debe actualizar al de handler. Entonces, el CORE recibe modificaciones internas, y las promueve como eventos de sincronía hacia el handler. El worker de sincronización toma estos eventos, evalúa el saldo en la base de datos y los impacta sobre el saldo del handler

IMPORTANTE: El saldo prioritario o mandatorio es el del CORE (base de datos Banksys) . Por esta razón existe el Worker de sincronización. La idea es que su misión sea la de empujar este valor hacia handler, para mantener sincronizado su saldo. El saldo de base es mandatorio, porque al impactar realmente el movimiento, surge el saldo definitivo que puede variar del impacto que hizo handler, ya que puede conllevar comisiones e impuestos. Además, la base de datos, esta dando servicio a otros canales que reciben movimientos.

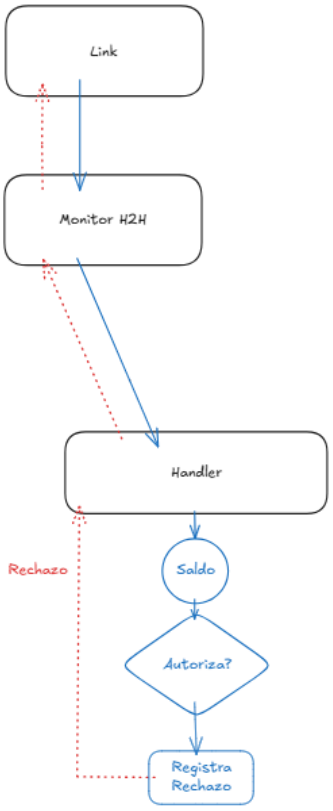
Además, serán necesarias modificaciones a aplicaciones del CORE para manejar comunicación de cambios en cuentas por canales internos.

Diseño Funcional

Tanto el handler de autorización como los workers de autorización, estan vinculados porque deben brindar servicio en forma coordinada, trabajan juntos. Para comprender su mecanismo de trabajo se presenta el siguiente diagrama presentando los 2 componentes y los momentos en que trabajan.

Handler de Autorización - Momento T - Caso de Rechazo de solicitud

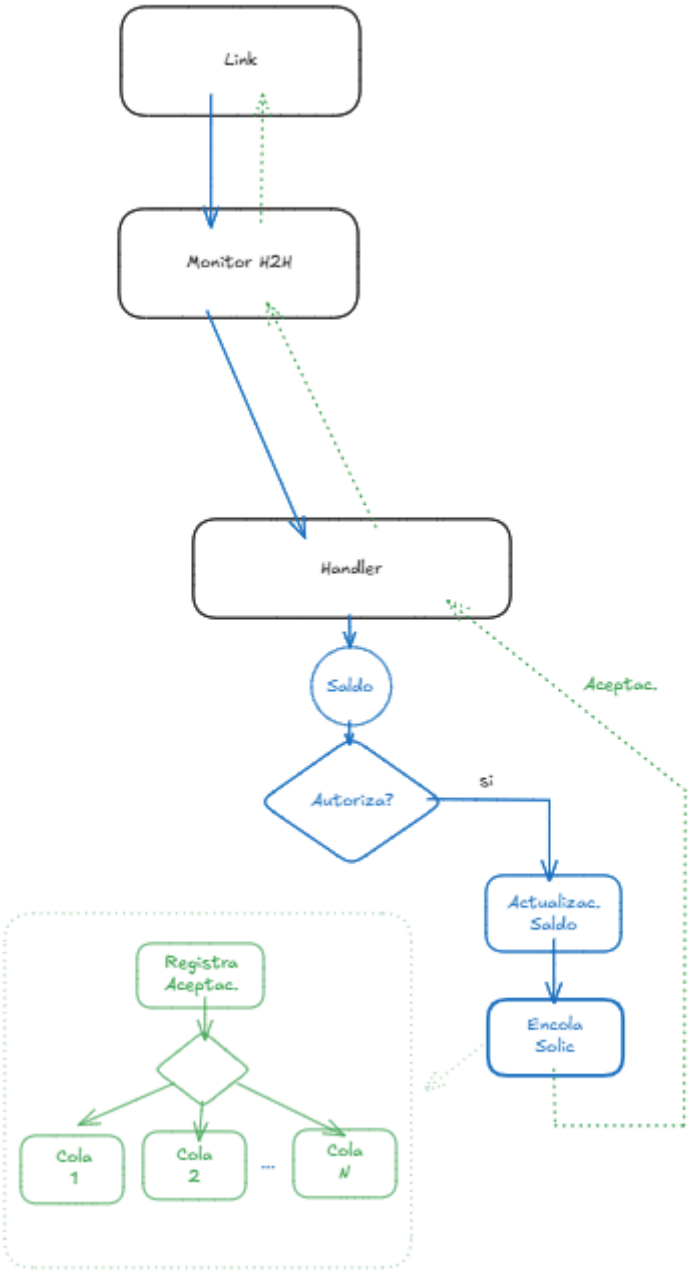
En el momento T, cuando llega una solicitud desde MONITOR, el caso mas sencillo, es cuando HANDLER evalua el saldo y por ser insuficiente rechaza el pedido:



El HANDLER recupera el saldo, lo compara con el monto solicitado, y si no es suficiente, registra la solicitud como rechazada y devuelve el resultado.

Handler de Autorización - Momento T - Caso de Autorización de solicitud

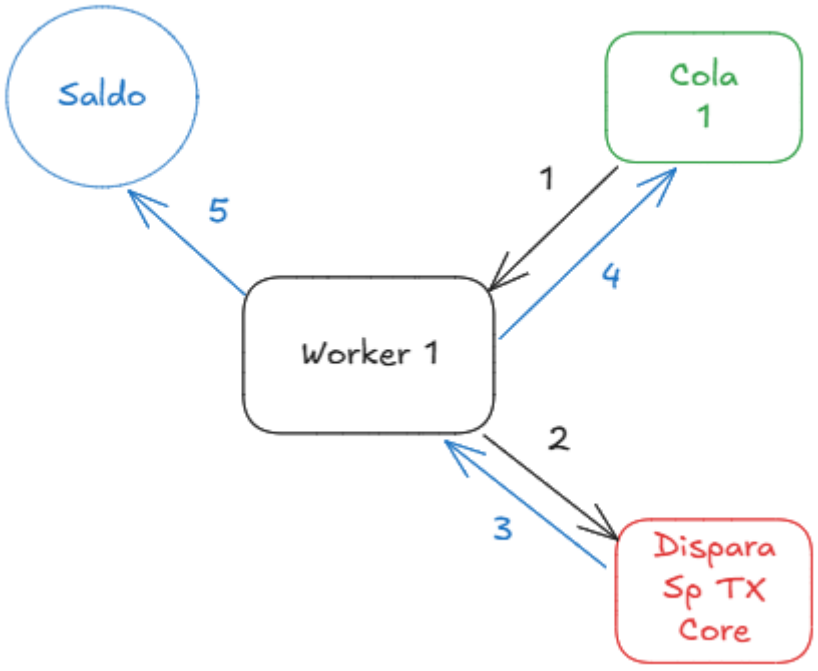
En el momento T, el caso que le sigue el complejidad, es cuando llega la solicitud desde el MONITOR y debe aceptar la misma. En este caso, HANDLER debe aceptar la misma, cambiar su saldo, y colocarla en la cola de solicitudes donde participa esa cuenta.



Cuando HANDLER encola la solicitud, no solo la registra, sino que define la cola en la que la solicitud será ingresada. En este paso, se tiene en consideración que las solicitudes de una misma cuenta se coloquen en la misma cola, para evitar condiciones de carrera.

Worker de solicitudes - Momento T+1

Los WORKERS de solicitudes, trabajan con las colas que se generan desde HANDLER. Se trata de un grupo, porque en realidad existe un WORKER por cada cola. Cada WORKER de solicitud, establece una conexión con la base de datos de HANDLER y con la base de datos del CORE. Esto es, para poder disparar las solicitudes y luego actualizar su estado.



El circuito de este WORKER es como sigue:

1 - WORKER Toma una solicitud de SU COLA DE SOLICITUD, con la información de la cuenta, el importe. Esta solicitud ya fue aprobada por HANDLER.
2 - WORKER Dispara la transacción en CORE. En este punto, CORE hace el impacto real en el saldo de Banksys de la transacción.
3 - WORKER recupera el resultado del envío a CORE.
4- WORKER actualiza el resultado en la solicitud
5- WORKER actualiza el saldo

Este circuito se repite por cada elemento de la cola.

La aplicación WORKER de solicitud debe poder ser instanciada, por cada cola que abra el HANDLER. Si un WORKER de solicitud se cae, el mismo debe poder ser “reemplazado” por otro, asignando la cola en su estado.

Diseño Técnico

Handler de autorización

- Tipo de aplicación
    - Se trata de una aplicación Web API .NET Core version 8, en esta instancia de prototipo
- Expone sus funciones por APIs
- Comunica eventos por un administrador de colas.
- Persistencia
    - Inicialmente SQL Server para repositorio interno. Como se usa ORM para almacenamiento se podrá modificar a otra base de datos a futuro.
- Integración
    - Entity Framework para almacenamiento interno.
    - ADO.net para disparo de procesos a la base Bansys
    - RabbitMQ para cola de eventos
- Implementación en 3 capas:
    - Presentación → Carpeta controller
    - Lógica de negocios → Carpeta service
    - Infrastructure → Carpeta Repositorio
        - Clases para manejo de cola
        - Clases para manejo de EF
        - Clases para manejo de Sps
- Modelos → Carpeta Model para diseños internos
- Utils → Carpeta Shared para ubicar las clases transversales
- Apis →
    - GET → Health (indica el estado de handler)
        - Iniciando (procesando cola)
        - Activo
        - Inactivo
    - POST → ActivarHandler
    - POST → InactivarHandler
    - POST → ConfigurarColas (establece el parametro de cantidad de colas)
    - GET → Estadística (indicadores de procesamiento, longitud de cola, tiempo promedio respuesta, etc)
    - GET → SaldoCuentaByCuenta (información de cuenta - saldo)
    - GET → SaldoCuentaAll (información de cuenta - saldo)
    - POST → ProcesarSolicitud (procesa el registro de una solicitud de débito)
    - POST → ProcesarCola (recarga la cola de eventos, con registros no procesados del repositorio interno, luego procesa los eventos)
    - PUT → ActualizarSaldo (actualiza el saldo del handler)
- Seguridad → JWT sin refresh token, inicialmente

Worker de solicitudes

- Tipo de aplicación
    - Se trata de una aplicación híbrida de servicio + web api .NET Core version 8, instanciable por cada cola que maneje HANDLER, en esta instancia de prototipo
- Expone funciones de activación y consulta estado por APIs.
- Ejecuta su función principal de procesamiento en segundo plano por medio de un Background process con cancelation token
- Persistencia → n0o posee propia, usa la base de datos de handler para actualización de respuestas (estado + saldos)
- Integración
    - Entity Framework para actualización en base de datos de handler
    - ADO.net para disparo de procesos a la base Bansys
    - Cliente RabbitMQ para acceder a eventos
- Implementación en 3 capas:
    - Presentación → Carpeta controller para apis
    - Lógica de negocios → Codigo worker
    - Infrastructure → Carpeta Repositorio
        - Clases para actualización del estados/saldos de handler
        - Clases para manejo de cola
        - Clases para manejo de Sps
- Modelos → Carpeta Model para diseños internos
- Utils → Carpeta Shared para ubicar las clases transversales
- Apis →
    - GET → Health (indica el estado de worker)
        - Activo
        - Inactivo
    - POST → ActivarWorker
    - POST → InactivarWorker
- Seguridad → JWT sin refresh token, inicialmente

Worker de Sincronización de saldos

- Tipo de aplicación
    - Se trata de una aplicación híbrida de servicio + web api .NET Core version 8, en esta instancia de prototipo
- Expone funciones de activación y consulta estado por APIs.
- Ejecuta su función principal de procesamiento en segundo plano por medio de un Background process con cancelation token
- Persistencia → no posee propia, usa la base de datos de handler para actualización de respuestas (estado + saldos)
- Integración
    - Entity Framework para actualización en base de datos de handler
    - ADO.net para disparo de procesos a la base Bansys
- Implementación en 3 capas:
    - Presentación → Carpeta controller para apis
    - Lógica de negocios → Código worker
    - Infrastructure → Carpeta Repositorio
        - Clases para actualización del estados/saldos de handler
        - Clases para manejo de cola
        - Clases para manejo de Sps
- Modelos → Carpeta Model para diseños internos
- Utils → Carpeta Shared para ubicar las clases transversales
- Apis →
    - GET → Health (indica el estado de worker)
        - Activo
        - Inactivo
    - POST → ActivarWorker
    - POST → InactivarWorker
- Seguridad → JWT sin refresh token, inicialmente