Sistema de Pagos Orientado a Eventos

Desafío Técnico Draftea - Agustín Pazos

Tabla de contenidos

Parte 1

Letra del problema

Diseñar una arquitectura integral de sistema de procesamiento de pagos orientada a eventos que maneja pagos de servicios, gestión de billeteras, recolección de métricas y respuestas de pago asincrónicas con manejo de errores.

Funcionalidad Principales:

- **Procesamiento de Pagos**: Los usuarios pueden pagar servicios usando el saldo de su billetera.
- **Gestión de Billetera**: Rastrear saldos de usuarios, deducir fondos, manejar reembolsos.
- **Recolección de Métricas**: Análisis en tiempo real de patrones de pago y tasas de éxito.
- **Procesamiento Asincrónico:** Manejar respuestas de pasarelas de pago externas.
- **Recuperación de errores**: Manejo integral de fallas y compensación.

Hipótesis de trabajo

En este punto se definen hipótesis de trabajo y las limitaciones a tener en cuenta para el diseño y desarrollo de la solución.

Billetera y Pagos

- Se asume que la solución utiliza una billetera custodiada, donde los fondos de esta billetera se utilizan para realizar los pagos de servicio. Por lo tanto, una vez que los fondos son reservados de la billetera, es la cuenta bancaria del sistema de pagos la que ejecuta la transacción final a través de la pasarela de pago, lo que asegura que la cuenta del usuario no esté directamente expuesta al proceso de la pasarela.

- El ingreso de dinero a la billetera del usuario está fuera del alcance del ejercicio.

- Para el ejercicio el sistema de pago maneja exclusivamente una única moneda, (USD) lo que simplifica enormemente la arquitectura al eliminar la complejidad de las conversiones de monedas y tarifas.

.

- Se asume que cada usuario posee una única billetera, lo que simplifica la arquitectura al no requerir la gestión de múltiples billeteras por usuario.

- Cada pago de servicio se considera una transacción externa que requiere la conexión con una pasarela de pago. La solución no contempla pagos de servicios internos, que se resolverían con una simple reducción de saldo en la billetera del usuario.

- Para limitar el alcance de la solución, cada pago de servicio se realiza por un monto fijo y predefinido, lo que elimina la posibilidad de que los usuarios indiquen el monto a reducir de sus billeteras.

- Los datos requeridos por la pasarela de pago varían según el país, para el alcance de este problema definimos una estructura genérica para la pasarela de pagos. Por lo tanto se define que para realizar un pago en la pasarela de pagos se requiere esta información:
    - número de cuenta de origen (cuenta interna de la empresa)
    - número de cuenta y código de banco de la cuenta destino (perteneciente al proveedor del servicio)

Gestión de pagos y dependencias externas

- La idempotencia es un requisito fundamental del sistema. Se utilizará el idempotency key generada por el cliente para garantizar que cada solicitud de pago sea procesada una única vez.

- La solución se integrará con una única pasarela de pago que gestionará todas las transacciones.

- No se requiere ningún tipo de comunicación con los proveedores de servicios que se intentan pagar después de la finalización del pago.

Comunicación y Notificaciones

- La notificación sobre el resultado de un pago exitoso o fallido será asíncrona.

- El alcance de este ejercicio se limita a la notificación de clientes móviles. La comunicación sobre el éxito o fallo de un pago se realizará exclusivamente a través de servicios de notificaciones push, como Firebase Cloud Messaging (FCM) o Apple Push Notification Service (APNs). La integración con otros canales como WebSockets para aplicaciones web o webhooks para clientes B2B no está incluida en esta etapa.

Servicios

- Queda por fuera del alcance del ejercicio entrar en detalle en los servicios de Auth, Usuario y de Órdenes. Se indican su propósito en el capítulo de componentes de la arquitectura, pero no se entrará en detalle en su análisis.

Análisis histórico de datos

- El análisis del comportamiento histórico del sistema, a través de la ingesta de eventos en un data warehouse o una base de datos de timeseries como InfluxDB, queda fuera del alcance del problema.

Concurrencia del Sistema

- Aunque el ejercicio no especifica una cantidad de transacciones de pagos por segundo, la arquitectura fue diseñada con la escalabilidad horizontal en mente como principal requerimiento, para manejar picos de tráfico y un crecimiento futuro del volumen de pagos.

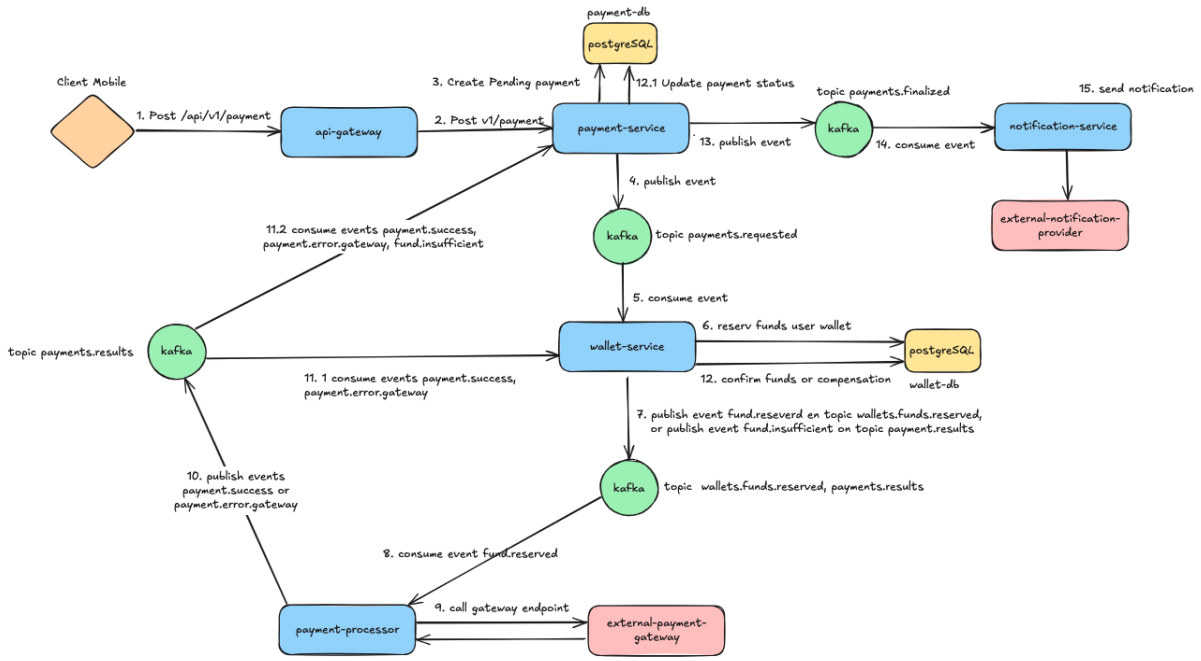
Seguridad

- En este ejercicio, la seguridad se enfoca en el API Gateway como el punto de control principal. Se asume que este componente es el responsable de la seguridad de las solicitudes, manejando la autenticación y autorización mediante la validación de tokens JWT. Su propósito es asegurar que solo las peticiones válidas y autorizadas lleguen a los servicios de pago.

- Este análisis no incluye la seguridad a nivel de red o de infraestructura. Componentes de seguridad perimetral como los **Web Application Firewalls (WAFs)** y las configuraciones de seguridad de los **Load Balancers**, dejando fuera del alcance la mitigación de ataques de denegación de servicio (DDoS), etc.

Diagrama de Arquitectura del Sistema

Este diagrama de arquitectura presenta una visión global del sistema de pagos, se indican sus principales componentes y el flujo de interacción entre los mismos. El diseño se basa en una arquitectura de microservicios, con un enfoque en el desacoplamiento, la resiliencia y la escalabilidad.



Componentes de la arquitectura

En este capítulo se profundiza la arquitectura del sistema de pagos, detallando cada uno de sus componentes clave. Para cada microservicio, se ofrece una descripción detallada de su propósito, sus responsabilidades y su rol dentro del flujo de eventos.

Principales componentes de sistema:

API Gateway: El único punto de entrada para todas las solicitudes de los clientes. Se encarga de la autenticación, la autorización y el ruteo al microservicio correcto, en el caso de la solicitud de pago al Payment Service.

Payment Service: El orquestador del proceso de pago. Es la única fuente de la verdad para el estado de un pago y coordina el flujo del proceso de pago.

Wallet Service: Gestiona los fondos del usuario. Su principal responsabilidad es verificar y reservar los fondos para un pago y, finalmente, debitar el monto final.

Payment Processor: El servicio que se comunica con la pasarela de pago externa. Maneja la llamada sincrónica para procesar el pago y traduce la respuesta de la pasarela en un evento para el resto del sistema.

.

Notification Service: Un servicio dedicado a notificar al usuario sobre el resultado de un pago. Consume eventos y envía notificaciones a través del canal adecuado para el cliente (por ejemplo, notificaciones push móviles).

Kafka: Facilita la comunicación asíncrona entre todos los microservicios, asegurando que el sistema esté desacoplado y sea resiliente.

Bases de Datos: Los servicios de Payment Service y Wallet Service tiene su propia base de datos, par**a** mantener la integridad de los datos y su independencia.

Pasarela de pago externa: Un servicio de terceros que procesa la transacción financiera.

Auth Service: Servicio utilizado para generar, validar JWT token, decodificación del JWT token. El Auth service se utiliza desde el API Gateway.

User Service: Servicio que tiene la responsabilidad de gestionar la información del usuario, se puede consultar si un usuario existe y la información de configuración para envío de notificaciones al dispositivo móvil del usuario. Este servicio se invoca desde el Notification Service.

Order Service: Servicio que tiene la responsabilidad de tener la información de las órdenes de servicio pendientes de pago por el usuario. El usuario cuando realiza un pago únicamente tiene que indicar el id de orden de servicio a pagar y a partir de esto se obtienen los datos asociados al pago pre registrados en el sistema.

Detalle de componente

API Gateway

El API Gateway es el punto de entrada unificado y centralizado para todas las solicitudes del cliente. Su propósito principal es actuar como un proxy inverso y una capa de seguridad, abstrayendo la complejidad de la arquitectura de microservicios para el cliente.

Propósito y Responsabilidad:

- **Autenticación y Autorización:** Valida los tokens JWT en las solicitudes entrantes, y lo inyecta en el encabezado de la solicitud para su uso por los servicios internos.
- **Ruteo:** Dirige las solicitudes del cliente al microservicio correspondiente, en este ejercicio al Payment Service.
- **Rate limit**.
- **Modificación de solicitud**: Agregar el user_id decodificado del JWT token en el body de la request.

Tecnología y stack:

- AWS API Gateway.

API (Endpoints):

- POST /api/v1/payments
- POST /api/v1/payments/{:payment_id}

Flujo de eventos:

- Comunicación sincrónica HTTPS entre cliente y servicio interno (Payment Service).

Dependencias:

- **Payment Service:** Servicio que gestiona las solicitudes de pago.
- **Auth Service:** servicio que valida y decodifica JWT tokens.

Escalabilidad:

- El componente API Gateway es sin estado (stateless), permite escalamiento horizontal, agregando varias instancias detrás de un balanceador de carga para distribuir el tráfico.

Métricas y observabilidad:

- Las métricas se exponen a través de un endpoint /metrics en un formato compatible con Prometheus. Estas métricas se utilizan para crear dashboards de monitoreo que visualizan el rendimiento y la salud del componente, y además para configurar alertas que notifican al equipo sobre problemas en el comportamiento del componente.
- Principales métricas:
    - Total de peticiones por segundo.
    - Latencia de peticiones.
    - Total de respuesta de peticiones por status code (2xx, 4xx, 5xx)
    - Total de conexiones activas.

Payment Service

El microservicio Payment Service es el orquestador de pagos, actúa como una única fuente de verdad sobre el estado de un pago, coordina flujo de eventos que finaliza con un pago exitoso o fallo.

Propósito y Responsabilidad:

- **Orquestación de eventos de pago:** Inicia flujo de pagos al recibir una request.
- **Gestión de estado de pagos**: Mantiene el estado de pago actualizando la base de datos de payments.
- **Validación de idempotencia:** Válida la de la request idempotency key para evitar pagos duplicados.

Tecnología y stack:

- Golang

API (Endpoints):

- Recibe las solicitudes de pago del API Gateway. **POST /v1/payments**
- Permite a otros clientes consultar el estado de los pagos.
  GET /v1/payments/{:payment_id}

Flujo de eventos:

- **Pública:** pública evento en tópico **payments.requested** (inicia el flujo de la Saga) y **payments.finalized** (notifica el estado final del pago).
- **Consume:** consume eventos payment.success, payment.error.gateway y fund.insufficient del topico **payments.results**.

Dependencias:

- **Kafka**: Para la comunicación asíncrona.
- **PostgreSQL**: Para la persistencia del estado del pago.

Patrones de diseño:

- Patrón de orquestación Saga
- Patrón transactional Outbox
- Distributed Tracing

Escalabilidad:

- El servicio es **sin estado (stateless)** en el contexto de la solicitud HTTP, lo que permite escalar horizontalmente con facilidad.
- El patrón transactional **Outbox** garantiza que las transacciones sean resistentes a fallos y no impidan el procesamiento de nuevas solicitudes.

Métricas y observabilidad:

- Las métricas se exponen a través de un endpoint /metrics en un formato compatible con Prometheus. Estas métricas se utilizan para crear dashboards de monitoreo que visualizan el rendimiento y la salud del componente, y además para configurar

alertas que notifican al equipo sobre problemas en el comportamiento del componente.
- Principales métricas:
    - **Latencia de API**: Mide el tiempo de respuesta de tus endpoints, como POST /v1/payments. Divide las métricas por endpoint y estado de respuesta.
    - **Tasa de Pagos**: Rastrea el total de pagos procesados. Etiqueta esta métrica por el estado del pago.
    - **Métricas Kafka:** número de mensajes publicados y consumidos.
    - **Métricas de Postgres:** incluyendo la latencia de las consultas y el uso de conexiones.

Wallet Service

Es el dueño de los fondos de la billetera del usuario. Funciona como la única fuente de verdad para los datos de la billetera. Su principal responsabilidad es garantizar la integridad de los saldos de la billetera.

Propósito y Responsabilidad:
- **Gestión de Saldos**: Mantiene el registro de los saldos de los usuarios.
- **Reserva y Débito de Fondos**: En un proceso de dos pasos, reserva los fondos para un pago y luego los debita permanentemente al confirmarse la transacción.
- **Auditoría de Transacciones**: Registra un historial de todos los movimientos de fondos en las billeteras para fines de auditoría y conciliación.

Tecnología y stack:
- Golang

API (Endpoints):
- Este microservicio opera de forma asíncrona consumiendo eventos de Kafka. Pero fuera de este ejercicio se podría evaluar crear un endpoint de lectura para que otros microservicios consulten el saldo de una billetera.

Flujo de eventos:
- **Pública**: Pública evento **fund.reserved** en el tópico de **wallets.funds.reserved** (si la reserva es exitosa) o evento **fund.insufficient** en tópico **payments.results** si el saldo es insuficiente.
- **Consume**: consume evento en tópico **payments.requested** para iniciar la reserva de fondos. También consume los eventos payment.success, payment.error.gateway del tópico **payments.results** para debitar permanentemente o liberar el saldo previamente reservado.

Dependencias:
- **Kafka**: Para comunicación asíncrona de publicación de eventos y consumir eventos de los microservicios.
- **PostgreSQL**: Para el almacenamiento persistente de saldos y el historial de transacciones.

Patrones de diseño:

- Patrón de orquestación Saga. Realiza un paso (reservar/debitar fondos) en la transacción distribuida iniciada por payment service.
- Patrón transactional Outbox.

Escalabilidad:

- El servicio es sin estado (stateless) con respecto a la sesión de un usuario, lo que permite la escalabilidad horizontal. Múltiples instancias pueden consumir del mismo tópico de Kafka, distribuyendo la carga de procesamiento.

Métricas y observabilidad:

- Las métricas se exponen a través de un endpoint /metrics en un formato compatible con Prometheus. Estas métricas se utilizan para crear dashboards de monitoreo que visualizan el rendimiento y la salud del componente, y además para configurar alertas que notifican al equipo sobre problemas en el comportamiento del componente.
- Principales métricas:
    - Número de fondos reservados.
    - Número de débitos confirmados.
    - Número de fondos insuficientes.
    - **Métricas Kafka:** número de mensajes publicados y consumidos.

Payment Processor

El Payment Processor es un componente sin estado (stateless) y adaptador de la pasarela de pagos externa sincronica. Es el único servicio en el sistema que tiene el conocimiento de la API de pasarela de terceros.

Propósito y Responsabilidad:

- El payment processor tiene la responsabilidad de hacer las llamadas sincrónicas a la pasarela de pagos para realizar el pago del servicio.
- Adaptador de la respuesta de la pasarela de pagos al formato de eventos manejados en el sistema de pago interno.
- Resiliencia en llamado a la pasarela de pagos externa, debe de manejar lógica de reintentos para gestionar fallos temporales en la pasarela de pagos y uso de patron circuit breaker.

Tecnología y stack:

- Golang

API (Endpoints):

- Este servicio no expone API, funciona como un consumidor de eventos.

Flujo de eventos:

- **Consume**: consume evento **fund.reserved** del tópico **wallets.funds.reserved** (activado por el Wallet Service una vez que ha asegurado los fondos. Este es el evento que le indica al Payment Processor que puede iniciar la llamada a la pasarela).
- **Pública:** eventos **payment.success** o **payment.error.gateway** dependiendo si el pago fue exitoso o no en el tópico **payments.result**.

Dependencias:

- Kafka
- Pasarela de pago externa.
- PostgreSQL: Para la persistencia de respuesta del gateway.

Patrones de diseño:

- Lógica de Reintentos: Aplica una política de reintentos con backoff exponencial para fallos temporales de la pasarela.
- Circuit Breaker
- Transactional Outbox

Escalabilidad:

- El componente payment processor es sin estado (stateless) y procesa de forma asíncrona, se pueden agregar instancias para manejar alta demanda de pagos.

Métricas y observabilidad:

- Las métricas se exponen a través de un endpoint /metrics en un formato compatible con Prometheus. Estas métricas se utilizan para crear dashboards de monitoreo que visualizan el rendimiento y la salud del componente, y además para configurar alertas que notifican al equipo sobre problemas en el comportamiento del componente.
- Principales métricas:
    - Latencia de la Pasarela, el tiempo que toma procesar una llamada a la pasarela externa.
    - Tasa de Éxito/Fallo de la Pasarela.
    - Número de reintentos
    - Métrica de consumidor de Kafka.

Notification Service

Componente sin estado (stateless) responsable de comunicar el resultado final del pago al usuario, su propósito es desacoplar la lógica de procesamiento del pago de la notificación del resultado del mismo.

Propósito y Responsabilidad:

- Dadas las hipótesis de trabajo, el Notification Service tiene la responsabilidad de rutear las notificaciones de pago exclusivamente a dispositivos móviles. El servicio debe seleccionar y enviar el mensaje al canal correspondiente, ya sea Android (a través de FCM) o iOS (a través de APNs). Desacoplar lógica de notificación del procesamiento del pago.
- Resiliencia en fallos en servicio de notificación (FCM, APNs) mediante lógica de reintentos.

Tecnología y stack:

- Golang

API (Endpoints):

- Este servicio no expone API. Su única forma de operación es asíncrona, consumiendo eventos del bus de mensajes.

Flujo de eventos:

- **Consume**: evento en tópico **payments.finalized** (el evento final que indica que el pago ha sido procesado por completo y su estado ha sido actualizado en la base de datos).

Dependencias:

- Kafka: para el consumo de eventos
- Servicios de notificación externos, Firebase Cloud Messaging (FCM) para Android, Apple Push Notification Service (APNs) para iOS.
- Acceso a User service para solicitar datos requeridos para la configuración de la notificación de un user_id.

Escalabilidad:

- Completamente sin estado (stateless) y opera de forma asíncrona, lo que permite una escalabilidad horizontal.
- El componente es tolerante a fallos, ya que si una instancia falla, las otras pueden continuar procesando los mensajes de la cola de Kafka.

Métricas y observabilidad:

- Las métricas se exponen a través de un endpoint /metrics en un formato compatible con Prometheus.
- Métricas de interés:
    - Tasa de notificaciones enviadas por canal.
    - Tasa de fallo de envío de notificación.
    - Latencia en envío de la notificación.
    - Métrica de consumidor de Kafka.

Catálogo de tópicos de eventos

En este capítulo se detalla el propósito de cada tópico definido, el servicio que lo publica y los servicios que lo consumen.

tópico payments.requested

Este tópico inicia el flujo de la saga de pagos. Representa la solicitud inicial para procesar un pago.

Publicador: Payment Service pública evento inmediatamente después de recibir una solicitud de pago del cliente y de registrar el pago en base de datos en estado `PENDING`.

Consumidor: Wallet Service lo consume para validar y reservar los fondos necesarios para el pago.

tópico wallets.funds.reserved

Un evento de éxito en el tópico **wallet.funds.reserved** indica que el Wallet Service ha validado que el usuario tiene fondos suficientes y ha procedido a reservarlos.

Publicador: Wallet Service pública evento *fund.reserved* si la reserva de fondo fue exitosa.

Consumidor: Payment Processor consume evento *fund.reserved* para realizar la llamada a la pasarela de pago externa.

tópico payments.results

Un evento en el tópico **payments.result** indica que un pago fue exitoso o existió un error.

Publicador:

- El Wallet Service pública evento de tipo *fund.insufficient* si el usuario no tiene fondos para realizar el pago en la billetera.

- El Payment Processor pública evento de tipo
- El Payment Processor pública evento de tipo *payment.error.gateway* cuando existe un error en la pasarela de pago externa.
- El Payment Processor pública evento de tipo *payment.success* cuando el pago fue realizado con éxito en la pasarela de pago.

Consumidor:

- El Wallet Service consume el tópico **payments.results** los eventos de tipo *payment.error.gateway o payment.success* para impactar en el saldo correcto los fondos de la billetera del usuario.
- El Payment Service consume el tópico **payments.results** los eventos de tipo *payment.error.gateway o payment.success o fund.insufficient* para impactar el pago en el estado correcto.

tópico payment.finalized

Este tópico consolida el resultado final de la saga de pagos, ya sea éxito o fracaso, y se utiliza para notificar al usuario. Es el último paso del proceso de pagos.

Publicador: Payment Service pública evento *payment.finalized* después de que ha actualizado la base de datos de pagos con el estado COMPLETED o FAILED.

Consumidor: Notification Service consume el evento y posteriormente inicia el proceso de enviar notificación del resultado del pago al cliente.

Flujo de datos

Flujo Cliente → API Gateway

El cliente envía una solicitud HTTPS para iniciar el pago.

Método: POST
Endpoint: /api/v1/payments
Headers:

- authorization: Bear <JWT token>
- content-type: application/json
- idempotency-key: e5f03f47-817c-4a39-9f1d-4f11b22e1a3c

Body(JSON):
{ "external_order_id": "a735e5d1-12a1-42e3-9f89-c4b6f12e84d9" }

El usuario envía el **idempotency-key** <UUID> v4 para evitar que el servidor procese cada reintento como un nuevo pago.

A nivel de request body, por la hipótesis de trabajo definida solo es necesario enviar el **external_order_id**, que básicamente identifica para un servicio y un usuario el pago que se debe de realizar, en base a este **external_order_id** se obtiene el order_id, monto, moneda, nombre de servicio. El **order_id** autoincremental en la base de datos no es expuesto al exterior.

Flujo API Gateway → Payment Services.

El API Gateway rutea la solicitud de pago al Payment Service.

Método: POST
Endpoint: /v1/payments
Headers:

- content-Type: application/json
- idempotency-key: e5f03f47-817c-4a39-9f1d-4f11b22e1a3c
- x-user-id: 12345

Body(JSON):

```
{
  "external_order_id": "a735e5d1-12a1-42e3-9f89-c4b6f12e84d9"
}
```

El API Gateway obtiene el **user_id** a partir de decodificar el JWT token.

Flujo Payment Service → Kafka (payments.requested)

Una vez que el Payment Service tiene los datos completos, los valida y publica el evento payments.requested en Kafka.

Payload (JSON):

```
{
  "event_id": "f3f2d9e1-c8a7-4b6a-8d1e-5f6c4b2a1d3c",
  "event_type": "payment.requested",
  "timestamp": "2025-08-23T19:35:00Z",
  "payment_id": "8f11b22e-131c-4b53-a8f6-508a68b72e54",
  "external_order_id": "a735e5d1-12a1-42e3-9f89-c4b6f12e84d9"
  "idempotency_key": "e5f03f47-817c-4a39-9f1d-4f11b22e1a3c",
  "paymen_data": {
      "amount": "100.50",
      "currency": "USD",
      "service_name": "Servicio A",
      "service_account_number": "1234567890",
      "service_bank_code": "BNK112",
   },
  "currency": "USD",
}
```

Descripción de los campos:

- **event_id:** Un identificador único para el evento.
- **event_type:** nombre del tipo de evento de negocio que se manda: payment.requested
- **timestamp:** La marca de tiempo en que el evento fue creado.
- **payment_id:** El identificador único de la transacción.
- **external_order_id:** El identificador del pedido al que pertenece el pago. Se incluye para mantener un vínculo entre el dominio de pagos y el de pedidos.
- **idempotency_key**: La clave que protege al sistema de solicitudes duplicadas. Se propaga desde la solicitud inicial del cliente para que los consumidores la utilicen si es necesario.
- **user_id:** El identificador del usuario que realiza el pago.

- **payment_data**: Este objeto agrupa todos los detalles del pago.
    - **amount**: El monto de la transacción.
    - **currency**: El código de la moneda del pago (USD).
    - **service_name**, **service_account_number**, **service_bank_code**: Estos campos representan el destino de la transferencia. Proporcionan la información de la cuenta externa a la cual se debe de realizar la transferencia.

Flujo Payment Service → Cliente

El Payment Service responde con un HTTP 202 (Accepted). El flujo sincrónico termina. Esta respuesta indica que el pago del servicio está en proceso, el resultado de pago exitoso o fallo se comunica de manera asíncrona por el proceso de notificación.

Flujo Wallet Service → Kafka (funds.reserved)

Una vez que el wallet service reserva los fondos pública evento de funds.reserved en el tópico de wallets.fund.reserved

Payload (JSON):

```
{
  "event_id": "abcde123-4567-89ab-cdef-0123456789ab",
  "event_type": "funds.reserved",
  "timestamp": "2025-08-24T19:40:00Z",
  "payment_id": "8f11b22e-131c-4b53-a8f6-508a68b72e54",
  "idempotency_key": "e5f03f47-817c-4a39-9f1d-4f11b22e1a3c",
  "user_id": "12345",
  “paymen_data": {
      "amount": "100.50",
      "currency": "USD",
      "service_name": "Servicio A",
      "service_account_number": "1234567890",
      "service_bank_code": "BNK112",
   },
  "wallet_transaction_id": "3f2b1d7e-9c02-4d51-8b2a-0a7c9d6f3e1b"
}
```

Descripción de los campos:

- **event_id:** Identificador único del evento para propósitos de trazabilidad.
- **event_type:** nombre del tipo de evento de negocio que se manda: "funds.reserved".
- **timestamp:** La marca de tiempo de cuándo ocurrió el evento.

- **payment_id:** El identificador único de la transacción de pago.
- **idempotency_key**: La clave que protege al sistema de solicitudes duplicadas. Se propaga desde la solicitud inicial del cliente para que los consumidores la utilicen si es necesario.
- **user_id:** Identificador del usuario.
- **payment_data**: Este objeto agrupa todos los detalles del pago.
    - **amount**: El monto de la transacción.
    - **currency**: El código de la moneda del pago (USD).
    - **service_name**, **service_account_number**, **service_bank_code**: Estos campos representan el destino de la transferencia. Proporcionan la información de la cuenta externa a la cual se debe de realizar la transferencia.
- **wallet_transaction_id:** Un identificador único generado por el Wallet Service para su propia transacción de reserva. Este campo es crucial para el seguimiento y la posterior confirmación o reversión de los fondos.

Flujo Wallet Service → Kafka (fund.insufficient)

Este flujo se activa cuando el **Wallet Service** recibe una solicitud de reserva de fondos para un usuario, pero determina que el saldo disponible en la billetera es insuficiente. En lugar de procesar la transacción, el servicio publica un evento de fallo.

Payload (JSON):


```
{
  "event_id": "d8c7b6a5-f4e3-d2c1-b0a9-8d7c6b5a4f3e",
  "event_type": "funds.insufficient",
  "timestamp": "2025-08-24T19:35:05Z",
  "payment_id": "8f11b22e-131c-4b53-a8f6-508a68b72e54",
  "idempotency_key": "e5f03f47-817c-4a39-9f1d-4f11b22e1a3c",
  "user_id": "12345",
  “paymen_data": {
      "amount": "100.50",
      "currency": "USD",
      "service_name": "Servicio A",
      "service_account_number": "1234567890",
      "service_bank_code": "BNK112",
   },
  "reason": "insufficient_funds"
}
```


Descripción de los campos:

- **event_id:** Un identificador único para el evento que ayuda en la trazabilidad.
- **event_type:** Describe la acción de negocio: "funds.insufficient".
- **timestamp:** La marca de tiempo de cuándo ocurrió el evento.

- **payment_id:** El identificador único de la transacción de pago. Actúa como la clave de partición.
- **idempotency_key**: La clave que protege al sistema de solicitudes duplicadas. Se propaga desde la solicitud inicial del cliente para que los consumidores la utilicen si es necesario.
- **user_id:** Identificador del usuario.
- **amount y currency:** La información financiera del pago que no pudo completarse.
- **reason:** Este campo especifica la causa del fallo a nivel de negocio.

Flujo Payment Processor → Payment Gateway externo.

En la hipótesis de este ejercicio, se asume que el sistema ya ha recibido el dinero del usuario (debitado de la billetera) y ahora el Payment Processor realiza un pago a un tercero utilizando una cuenta bancaria o cuenta interna de la empresa.

El Payment Processor envía una solicitud HTTP POST a la pasarela de pagos para ejecutar la transferencia de fondos.

Solicitud a Payment Gateway

Método: POST
Endpoint: /v1/payments
Headers:
- Content-Type: application/json
- Idempotency-Key: <payment_id>

Body (JSON):

```
{
  "payment_id": "8f11b22e-131c-4b53-a8f6-508a68b72e54",
  "timestamp": "2025-08-24T19:35:05Z",
  "amount": "100.50",
  "currency": "USD",
  "source": {
    "account_number": "2143658709",
  },
  "destination": {
    "name": "Servicio A",
    "account_number": "1234567890",
    "bank_code": "BNK112",
  }
}
```

Descripción de los campos:

- **payment_id:** El identificador único de la transacción de pago
- **timestamp**: La marca de tiempo que indica cuándo se creó la solicitud
- **amount**: monto del pago del servicio
- **currency**: moneda del pago del servicio.
- **source**: En la estructura de source se indica en campo **account_number** con la cuenta de la empresa que paga el servicio.
- **destination**: En la estructura de destination se indica el campo **name**, **account_number** y **bank_code** de la cuenta del proveedor del servicio.

Respuesta exitosa

HTTP Status code: 200 OK


```
{
  "status": "approved",
  "message": "Payment processed successfully.",
  "gateway_transaction_id": "a735e5d1-12a1-42e3-9f89-c4b6f12e84d9",
  "payment_id": "8f11b22e-131c-4b53-a8f6-508a68b72e54"
}
```


Descripción de los campos:

- **status:** Indica que el estado de la transacción se procesó correctamente.
- **message:** mensaje descriptivo.
- **gateway_transaction_id**: Identificador único de la transacción a nivel de la pasarela de pagos
- **payment_id:** El identificador único de la transacción de pago. El que se envía en la request.

Respuesta HTTP Status code: 400 Bad Request


```
{
  "status": "failed",
  "error_code": "invalid_account_number",
  "message": "The destination account number is invalid.",
  "payment_id": "8f11b22e-131c-4b53-a8f6-508a68b72e54"
}
```


Descripción de los campos:

- **status:** Indica que la solicitud falló
- **error_code:** Código de error del Gateway.
- **message:** mensaje descriptivo del error
- **payment_id:** El identificador único de la transacción de pago. El que se envía en la request.

Respuesta HTTP Status code: 500 Internal Server Error


```
{
  "status": "failed",
  "error_code": "gateway_unailable",
  "message": "The payment gateway is temporarily unavailable",
  "payment_id": "8f11b22e-131c-4b53-a8f6-508a68b72e54"
}
```


Descripción de los campos:

- **status:** Indica que la solicitud falló temporalmente.
- **error_code:** Código de error del Gateway.
- **message:** mensaje descriptivo del error
- **payment_id:** El identificador único de la transacción de pago. El que se envía en la request.

Flujo Payment Processor → Kafka (payment.success)

Este flujo es donde un pago externo exitoso se convierte en un evento interno estandarizado para nuestro sistema.. El Payment Processor toma la respuesta de éxito de la pasarela y la normaliza en el evento payment.success de Kafka, que se envía al tópico payments.results.

Payload (JSON):


```
{
  "event_id": "b9e8f7d6-a5c4-3b2d-1a0f-9e8d7c6b5a41",
  "event_type": "payment.completed",
  "timestamp": "2025-08-24T19:35:15Z",
  "payment_id": "8f11b22e-131c-4b53-a8f6-508a68b72e54",
  "user_id": "12345",
  "amount": "100.50",
  "currency": "USD",
  "gateway_response": {
     "status": "approved",
     "message": "Payment processed successfully.",
     "gateway_transaction_id": "a735e5d1-12a1-42e3-9f89-c4b6f12e84d9",
  }
}
```

Descripción de los campos:

- **event_id**: Un identificador único para el evento, usado para trazabilidad.
- **event_type**: Describe la acción de negocio: payment.completed.
- **timestamp**: La marca de tiempo de cuándo ocurrió el evento.
- **payment_id**: El identificador de la transacción que actúa como la clave de partición de Kafka, garantizando que los eventos relacionados con este pago sean procesados en orden.
- **user_id**: El identificador del usuario.
- **amount y currency**: El monto y la moneda de la transacción que se completó exitosamente.
- **gateway_response:** Un objeto que encapsula la respuesta de la pasarela externa.
    - **status:** Indica que el estado de la transacción se procesó correctamente.
    - **message:** mensaje descriptivo.
    - **gateway_transaction_id**: Identificador único de la transacción a nivel de la pasarela de pagos

Flujo Payment Processor → Kafka (payment.error.gateway)

Este flujo se activa cuando el Payment Processor, recibe una respuesta de fallo de la pasarela de pago externa.

Payload (JSON):


```
{
  "event_id": "c4d3e2f1-b0a9-8c7d-6b5a-4e3d2c1b0a98",
  "event_type": "payment.failed.gateway_error",
  "timestamp": "2025-08-24T19:35:16Z",
  "payment_id": "pay-xyz789",
  "user_id": "usr-12345",
  "amount": "100.50",
  "currency": "USD",
  "gateway_response": {
      "status": "failed",
       "error_code": "invalid_account_number",
       "message": "The destination account number is invalid."
  }
}
```


Descripción de los Campos:

- **event_id**: Identificador único del evento para trazabilidad.
- **event_type**: Describe la acción de negocio: payment.failed.gateway_error.
- **timestamp**: La marca de tiempo del evento.
- **payment_id**: El identificador de la transacción que sirve como la clave de partición en Kafka.

- **user_id**: El identificador del usuario.
- **amount y currency**: El monto y la moneda de la transacción que falló.
- **gateway_response:** Un objeto que encapsula la respuesta de la pasarela externa
    - **status:** Indica que la solicitud falló.
    - **error_code:** Código de error del Gateway.
    - **message:** mensaje descriptivo del error.

Flujo Payment Service → Kafka (payment.finalized)

El payload del evento payment.finalized contiene los detalles esenciales que necesita el Notification Service para informar al usuario.

Payload (JSON):


```
{
  "event_id": "uvwxy456-a1b2-c3d4-e5f6-a1b2c3d4e5f6",
  "event_type": "payment.finalized",
  "timestamp": "2025-08-24T19:40:00Z",
  "payment_id": "8f11b22e-131c-4b53-a8f6-508a68b72e54",
  "user_id": "12345",
  "amount": "100.50",
  "currency": "USD",
  "status": "COMPLETED",
}
```

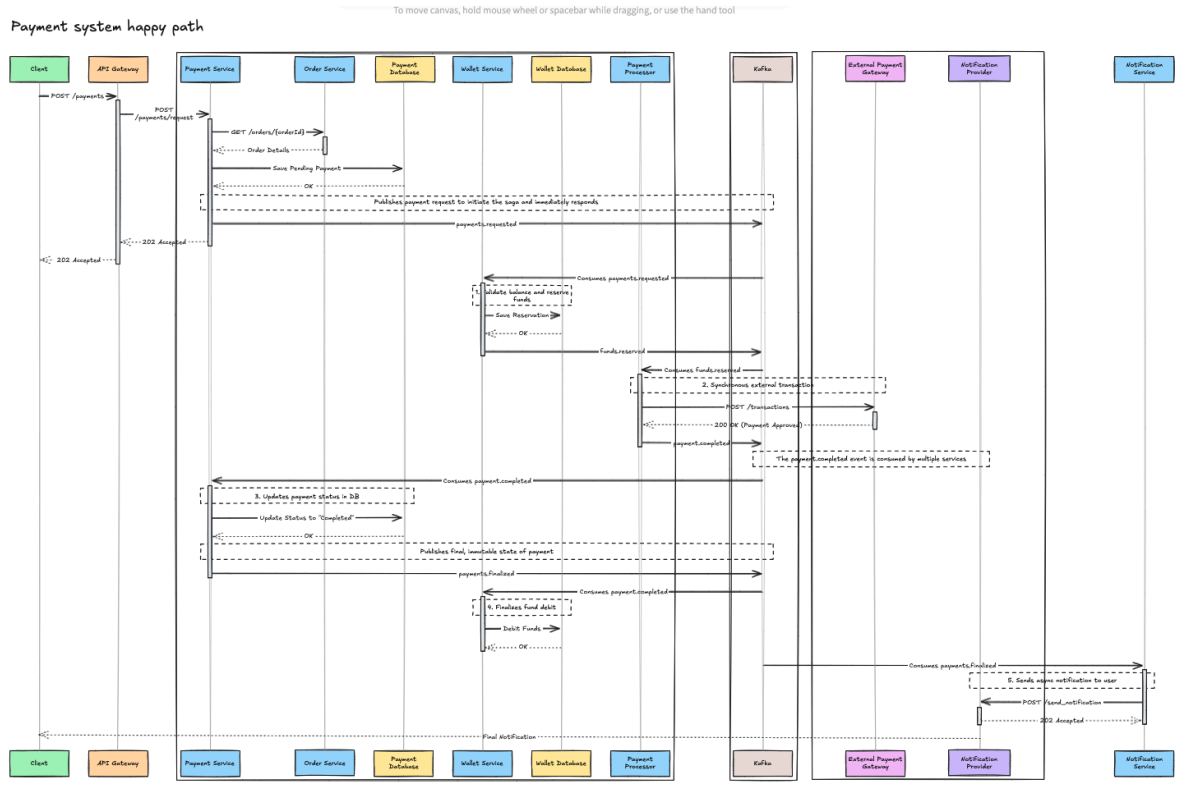

Descripción de los Campos:

- **event_id**: Un identificador único para el evento, útil para la trazabilidad.
- **event_type**: Describe la acción de negocio: payment.finalized.
- **timestamp**: La marca de tiempo en que el evento fue creado.
- **payment_id**: El identificador de la transacción que sirve como la clave de partición en Kafka para mantener la coherencia.
- **user_id**: El identificador del usuario que realizó el pago.
- **amount y currency**: El monto y la moneda de la transacción.
- **status**: Este campo indica el estado final del pago, ya sea "COMPLETED" o "FAILED".

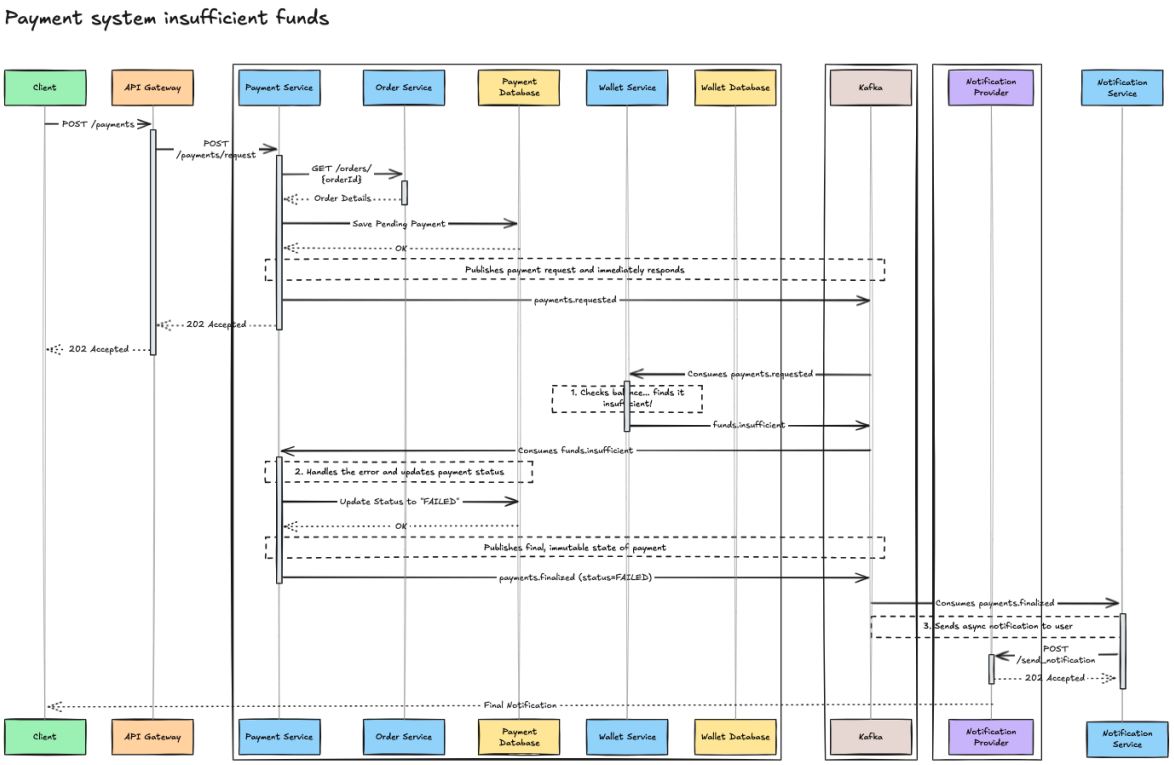
Análisis de distintos escenarios

Los diagramas presentados en este capítulo también están disponibles directamente desde excalidraw.

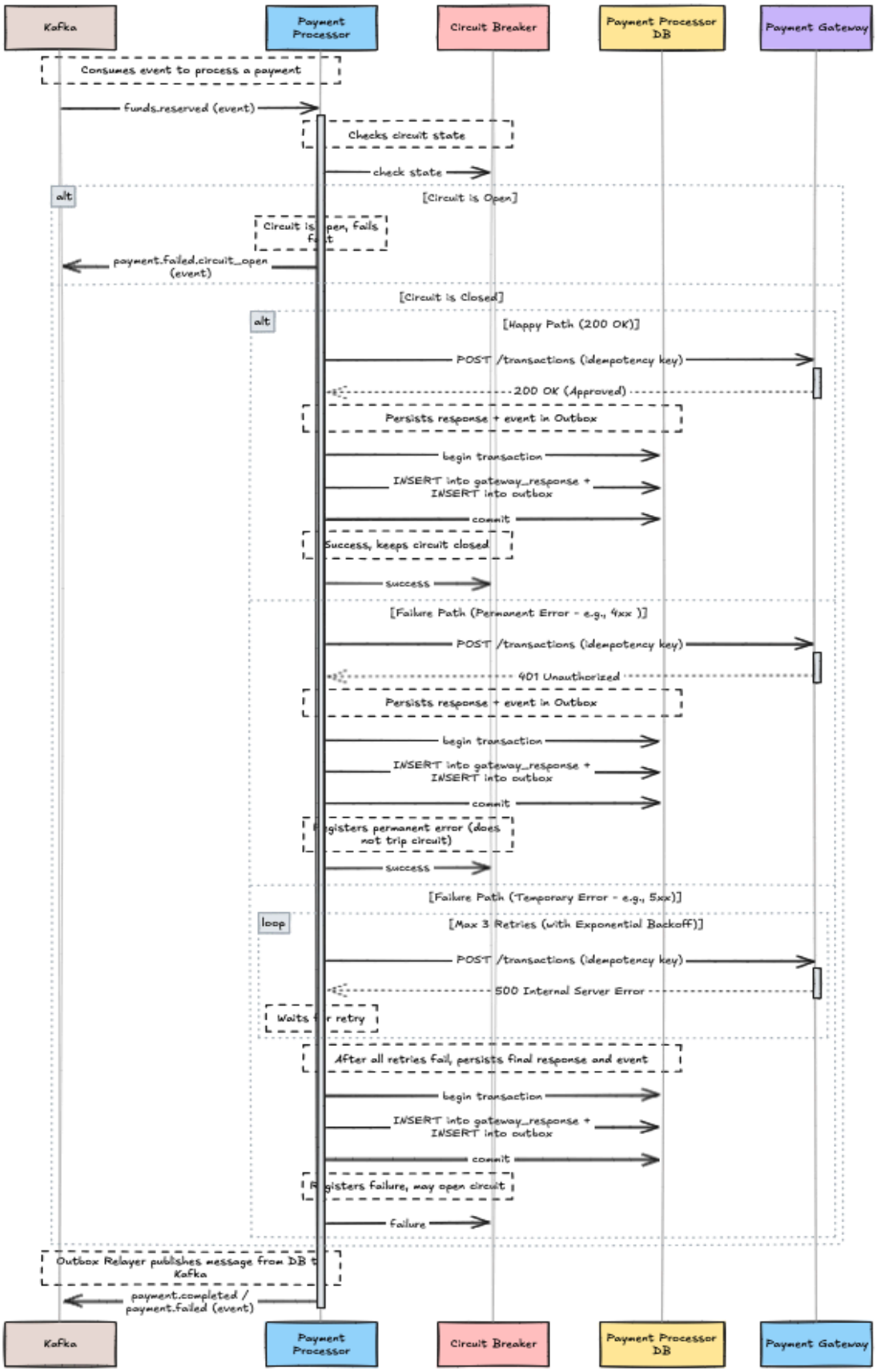
Pago exitoso



Saldo Insuficiente



Tiempo de Espera de Pasarela Externa: Reintento y fallback.



Pagos Concurrentes: Manejo de condiciones de carrera.

El Payment Service recibe una solicitud con una clave de idempotencia única. Esta clave sirve como primera línea de defensa para el Payment Processor, que evita intentos de pago duplicados. Al mismo tiempo, el Wallet Service usa un bloqueo distribuido con Redis para garantizar que solo una operación a la vez pueda modificar el saldo de una billetera, previniendo condiciones de carrera y sobregiros.

Recuperación del Sistema: Reinicio y reconstrucción del estado.

El sistema está diseñado para ser sin estado (stateless) y resiliente, lo que simplifica enormemente la recuperación después de un fallo.

Reinicio del Sistema: Si un microservicio de la arquitectura (como el Payment Processor o el Wallet Service) falla o se reinicia, no pierde su estado. Se reinicia y comienza a consumir eventos desde el último punto registrado en el log de Kafka. No necesita ningún tipo de recuperación de estado compleja, ya que la lógica de negocio se aplica a los eventos entrantes en tiempo real.

Reconstrucción del Estado: El estado del sistema se puede reconstruir en cualquier momento a partir del log de eventos inmutable de Kafka. Si, por ejemplo, la base de datos de un servicio se corrompe, se podría reconstruir por completo consumiendo todos los eventos desde el principio del tópico.

Parte 2

Alcance

Por letra hay una dedicación aproximada a esta tarea de 4 horas. Por lo tanto la mayor parte del tiempo se dedicó a nivel de código, donde este tiempo no es suficiente para la implementación de la solución, en este tiempo se logró avanzar en el componente de Payment Service para procesar un solicitud de pago y persistir el pago en base de datos y en la tabla outbox del outbox transaction pattern. Se inició también la creación de consumer de la tabla de outbox para el envío del evento al tópico de payment.requested. El código está disponible en el repositorio: https://github.com/walker-16/payment-system

Stack Tecnológico

Infraestructura: AWS
Microservicios: Golang
API Gateway: AWS API Gateway
Message Broker: Apache Kafka gestionado por Amazon MSK
Base de datos: PostgreSQL con Amazon RDS
Cache: Amazon ElastiCache
Container: Docker
Registro de Containers: Amazon Elastic Container Registry (Amazon ECR)
Gestión de Contenedores: Kubernetes gestionado por Amazon EKS

Seleccioné AWS como proveedor de infraestructura debido a mi experiencia con su plataforma, que considero superior a la de Azure y Google Cloud. Esta elección se justifica, además, por la reconocida madurez, seguridad, extenso ecosistema de servicios, escalabilidad, resiliencia y alta disponibilidad que AWS ofrece

La elección de Golang como lenguaje de programación para los microservicios de payment, wallet, payment processor y notification se basa en su rendimiento, eficiencia, modelo de concurrencia y simplicidad de mantenimiento.

La elección de Kafka como message broker se basa en la adopción de una arquitectura de coreografía de eventos, donde los microservicios consumen eventos publicados por otros servicios. Para este escenario, Kafka es una opción ideal debido a su durabilidad, inmutabilidad y capacidad de auditoría.

Kafka almacena los mensajes en un log de eventos inmutable, lo que significa que los eventos no se eliminan al ser consumidos. Esto lo convierte en una fuente de verdad confiable para el sistema de pagos.

Una contra de la elección de Kafka, es la falta de soporte nativo para colas de mensajes no entregados (dead letter queue), por lo que su implementación debe ser manejada a nivel de consumidor en caso de usarse el patrón dead letter queue.

Escalabilidad

Escalabilidad a nivel de microservicio

- A nivel de microservicio, el sistema se ha diseñado para ser sin estado (stateless), lo que permite una escalabilidad horizontal. Esto, en conjunto con Kubernetes, asegura que un número de réplicas (pods) de cada servicio esté siempre en ejecución. El balanceo de carga se gestiona eficientemente entre estos pods, y la Escalabilidad Horizontal Automática (HPA) puede ser configurada para manejar picos de tráfico. Aunque nunca la he usado, Kubernetes Event-driven Autoscaler (KEDA) también podría investigarse como una opción para escalar los pods según el retraso a nivel de Kafka.

- También a nivel de microservicio se puede escalar verticalmente, asignando más recursos de CPU y memoria.

Escalabilidad a nivel de base de datos

- A nivel de base de datos utilizando PostgreSQL por AWS RDS se puede escalar verticalmente aumentando los recursos de la base de datos, aumentando el CPU, Memoria, almacenamiento o cantidad de conexiones a base de datos.

- A nivel de escalamiento horizontal se puede:
    - Configurar réplicas de la base de datos postgres para mejorar la distribución de carga en las solicitudes de lectura.
    - Particionamiento de tablas para distribuir la carga de trabajo en N tablas más chicas. Es importante definir una buena clave de partición.

Escalabilidad a nivel de caché (amazon ElastiCache)

- Soy partidario de usar caché cuando es necesario y no antes, potencialmente si se detecta que se está sobrecargando la db de lecturas.
- Posibles usos de caché en el sistema:
    - Para la verificación de pagos duplicados se podría usar un caché de idempotencia a nivel del Payment Service, posiblemente usando patrón cache aside.
    - O para gestionar la reserva de fondos de la wallet se podría usar el patrón caché para bloqueo distribuido.

- A nivel de la herramienta amazon ElastiCache se permite tanto escalamiento horizontal como vertical.

Decisiones del diseño del sistema

- Se optó para el diseño de la solución, una arquitectura de microservicio utilizando comunicación asíncrona. El flujo de un pago se convierte en una coreografía de eventos, con cada servicio reaccionando a un evento en lugar de ser llamado directamente.

- A nivel de notificación al usuario que el pago quedó realizado o existió un fallo, se optó por notificación asíncrona. De todas maneras es adecuado disponibilizar un endpoint de consulta del estado de un pago **GET api/v1/payments/{:payment_id}** que se pueda consultar por el estado de un pago.

- En el Payment Processor, el componente encargado de interactuar con la pasarela de pago, se implementa un esquema de reintentos con backoff exponencial para manejar las fallas temporales, complementado con el patrón circuit breaker para evitar sobrecargar la pasarela ante errores recurrentes en un periodo de tiempo. Si el circuito se encuentra abierto, el sistema responde de forma inmediata al usuario con un error de pago, cancelando la transacción sin enviar el evento a una DLQ, priorizando así una respuesta rápida y clara sobre el estado del pago en tiempo real.

- Para el diseño de la solución se evaluó la alternativa de que el Payment Service realizando llamadas síncronas al Wallet Service y luego al Payment Processor, pero se descartó por generar un alto acoplamiento y fragilidad en la cadena síncrona, donde una falla interrumpe todo el flujo. Además, este enfoque obligaba al Payment Service a asumir lógica compleja de reintentos y compensación. En su lugar, se adoptó una arquitectura más desacoplada y basada en eventos, que mejora la resiliencia y simplifica la evolución de los servicios.

- Al ir diseñando el sistema el sistema se me ocurrio que podria estar bueno tener un endpoint de troubleshooting en cada uno de los componentes para poder identificar errores por usuario o payment_id y poder crear una herramienta de troubleshooting qué fácilmente de una visión global de que paso con un pago en cada parte del sistema ante algún error no esperado.

- Se optó por usar kafka como message broker en cada parte del sistema, para tener una solución homogénea, se tendría que evaluar si entre el payment service y el notification service es necesario tener un kafka o se podria tener un queue SQS. Esto va a depender del tráfico que se tenga y también la posibilidad de tener más de un consumidor para el tópico de payment.finalized como podría ser un servicio de métricas de negocio que persista los pagos en una base de datos timeseries.

- Para las operaciones de doble escritura como son inserción en base de datos y publicación de evento en kafka se decidió utilizar el patrón outbox transactional para asegurar la atomicidad entre estas dos operaciones.

- A nivel de partición id de kafka se definió configurarla por payment_id de manera de asegurar el orden de evento del pago.

- En la solución se implementa distributed tracing para facilitar el diagnóstico de errores y el seguimiento de flujos entre microservicios. Cada componente adjunta de forma consistente en sus trazas de log el campo de payment-id,, lo que permite correlacionar eventos a través de distintos servicios. Esta práctica asegura una trazabilidad clara del ciclo de vida de cada pago y simplifica la detección de incidentes.

Estructura de proyecto

Este proyecto está organizado como un mono-repositorio de Go, diseñado para la construcción de un sistema de pagos distribuido. La arquitectura se basa en microservicios, donde cada uno es un módulo de Go independiente con su propio archivo go.mod.

La integración entre estos servicios se maneja a nivel de la raíz del proyecto a través del archivo go.work. Esto permite que el desarrollo, las pruebas y las dependencias de cada servicio se gestionen de manera aislada, mientras que el workspace de Go proporciona una forma sencilla de trabajar con todos los módulos de manera local. Este enfoque garantiza una clara separación de responsabilidades y facilita el desarrollo.

Con base en su repositorio, aquí tiene una descripción profesional de la estructura de su proyecto, lista para un archivo `README.md`.

Estructura del Proyecto

Este proyecto está organizado como un **mono-repositorio de Go**, siguiendo un diseño estándar para la construcción de un sistema de pagos distribuido basado en microservicios. La estructura garantiza una clara separación de responsabilidades, simplifica la gestión de dependencias y promueve la reutilización de código.

Directorios principales:

- **services**: Es donde se encuentran todos los microservicios. Cada servicio es un módulo de Go autocontenido con su propia lógica y dependencias internas.
- **pkg**: Este directorio es para paquetes compartidos y reutilizables. El código aquí está destinado a ser importado por múltiples servicios, lo que asegura la consistencia de las utilidades en todo el sistema.
- **docs**: Esta carpeta se dedica a la documentación técnica, incluyendo diagramas de arquitectura y secuencia de la solución.
- **deploy:** Directorio donde se va a encontrar los manifiestos de k8s para el deploy.

Observabilidad: Estrategia de Implementación

Para asegurar que el sistema sea monitoreable y auditable, se implementó una estrategia de observabilidad enfocada en tres pilares: Logs, Métricas y Trazas.

Logs y Trazas

Usar logs estructurados y tracing distribuido para correlacionar eventos a través de microservicios.

Implementación:

- Logs: Todos los servicios emiten logs en formato JSON. Se incluye payment_id.
- Tracing: El API Gateway genera un payment_id UUID para cada solicitud. Este ID se incluye en el payload de los eventos de Kafka, permitiendo reconstruir la secuencia completa de una transacción.

Métricas

Exponer métricas de rendimiento y negocio para su recolección por Prometheus.

Implementación:

- Instrumentación: Se usa la librería **ansrivas/fiberprometheus** para instrumentar automáticamente las API construidas con Fiber. Esto nos da métricas estándar de HTTP como latencia, solicitudes por segundo y códigos de estado.
- Métricas de Negocio: Se añadieron métricas personalizadas para capturar el estado del negocio. Por ejemplo, el Wallet Service expone métricas como *funds_reserved_total* y *funds_insufficient_total*. Esto permite monitorear la salud del negocio, no solo la de la infraestructura.