# Solución Prueba Técnica Backend

Tareas asignadas:

1. **Diseño de Base de Datos**

Para el diseño de base de datos, opté por un esquema relacional simple y eficiente que permita registrar cada transacción de pago procesada. Los datos que considero claves para cada operación son: el cliente, el monto, la forma de pago, la fecha y hora en que se realiza la transacción, el estado del proceso y la razón de un posible fallo.

La entidad creada corresponde a la clase PaymentTransaction que incluye los campos:

|  |  |
| --- | --- |
| * Id | Identificador único autogenerado por cada transacción. |
| * customerId | Referencia al ID del cliente que realiza el pago. |
| * amount | Monto de la transacción. |
| * paymentMethod | Tipo de método utilizado (ej. Paypal, tarjeta de crédito). |
| * createdAt | Fecha y hora en que se realizó la transacción. |
| * status | Estado final de la operación (éxito o fallo). |
| * failureReason | En caso de fallo, una descripción del motivo. |

Para la persistencia utilicé JPA/Hibernate y utilicé una base de datos H2 en memoria para las pruebas locales.

1. **Validar Entrada**

Para garantizar la integridad de los datos recibidos desde el frontend, implementé un mecanismo de validación en el punto de entrada de la aplicación.

Para lograr lo anterior, diseñé una clase llamada PaymentRequest, que actúa como DTO para encapsular la información enviada por el cliente. Dicha clase contiene los campos:

|  |  |
| --- | --- |
| * customerId | Obligatorio. |
| * amount | Obligatorio y validado para que se encuentre entre $0.01 y $1000. |
| * paymentMethod | Obligatorio y no puede ser una cadena vacía. |

Para lograr estas validaciones, utilicé anotaciones de jakarta.validation.constraints, como @NotNull, @DecimalMin, @DecimalMax y @NotBlank. De esta forma evito que se procesen datos incompletos o no válidos desde un inicio. Luego integré esta validación directamente en el controlador PaymentController, usando la anotación @Valid sobre el parámetro del método processPayment para que Spring valide la solicitud antes de ejecutar la lógica del servicio. y también agregué mensajes a cada restricción para que el frontend reciba las debidas respuestas de forma clara en caso de error.

1. **Procesar pago**

Esta tarea corresponde a la lógica de la orquestación del pago. La cual consiste en los siguientes pasos:

Recibir una solicitud válida

Enviar los datos a un servicio externo

Aplicar pago y registrar el resultado

Este proceso se encuentra centralizado en la clase de servicio llamada PaymentService. En este servicio seguí estos pasos:

1. Creación de la transacción -> Construcción del objeto PaymentTransaction.
2. Orquestación del pago -> cliente HTTP con OpenFeign para comunicación con otro servicio REST.
3. Manejo de Errores -> Si el servicio devuelve un resultado negativo se lanza una excepción.
4. Persistencia -> En todos los casos, se guarda la transacción en la BD por medio de PaymentTransactionRepository.
5. Respuesta estructurada -> Se devuelve una respuesta estructurada al frontend mediante el objeto PaymentResponse.
6. **Generar Respuestas**

Para la generación de las respuestas, definí una clase PaymentResponse que encapsula la información relevante del resultado de cada transacción. Esta clase incluye dos campos principales:

status: Refleja si la operación fue exitosa o fallida.

message: Contiene una descripción del resultado, ya sea confirmando el éxito del pago o explicando la razón del fallo.

**HINT: ¿Consideras enriquecer esos response?**

Considero que, al tratarse de una prueba técnica, no consideraría enriquecer el response, sin embargo, si se tratase de un sistema que está destinado a crecer o a conectarse con otros módulos, por ejemplo, un panel administrativo o facturación, sí sería recomendable enriquecer el response, proporcionando datos relevantes como el ID del cliente que realizó la transacción, el monto, el tipo o método de pago y la fecha de la misma. En ese caso modificaría la clase PaymentResponse para agregar esos campos adicionales, así como el servicio para retornarlos.

## Preguntas de seguimiento:

1. **¿Cómo diseñarías las pruebas unitarias para esta implementación para asegurar que se cubran todos los casos extremos?**

Para las pruebas unitarias, implementé Junit y mockito y para cubrir los casos extremos, diseñé pruebas unitarias en torno al componente PaymentService, ya que como mencioné previamente, contiene la lógica de negocio.

Simulé el comportamiento del cliente externo (ExternalPaymentClient) y del repositorio (PaymentTransactionRepository) usando Mockito.

Los casos que cubrí fueron los siguientes:

* Pago exitoso (respuesta success = true del servicio externo)
* Pago fallido (respuesta success = false con un mensaje de error)
* Excepción lanzada por el servicio externo (simulando una caída del ERP)
* Persistencia correcta de la transacción en todos los escenarios.

1. **¿Qué cambios recomendarías en una revisión de código para que esta solución esté lista para producción?**

Los cambios que recomendaría para que esta solución esté lista para producción durante una revisión de código serían:

* Enriquecer el response para incluir más detalles útiles de la transacción.
* Validar que el id del cliente exista en un sistema maestro.
* Que el tipo de transacción sea uno de los permitidos por la entidad (por medio de un enum o lista).
* Agregar un controlador centralizado para el manejo de errores de forma global con @ControllerAdvice.
* Agregar documentación de la API con Swagger.

1. **¿Cómo asegurarías que el sistema sea resiliente a fallos en el servicio central de pagos?**

Agregaría Resilience4j o similar, también se podrían integrar herramientas de monitoreo y alertas para detectar las caídas del ERP o un incremento en los tiempos de respuesta.

1. **¿Qué debería hacer la API que aplica el pago cuando recibe un error HTTP 500?**

Un error 500 indica un fallo interno en el servicio que aplica el pago, por lo tanto, es una señal clara de que hubo un fallo al momento de procesar el pago, en ese caso el orquestador debe manejar dicha excepción, por medio de un try-catch, por ejemplo. Como, para este caso utilicé Feign para realizar la petición al servicio externo, podría capturar un FeignException, marcar la transacción como fallida en la BD con un mensaje indicando el error y devolver un mensaje al frontend indicando que no se pudo completar la transacción.

1. **¿Cuál debería ser el comportamiento de la API cuando falla al procesar el pago?**

En primer lugar, se debe registrar la transacción con estado FAILED, guardar la causa del fallo y devolver al frontend un 200 OK con un BODY que indique claramente que el pago no fue procesado.

1. **¿Cómo debería manejar el sistema los reintentos si el servicio central no está disponible temporalmente?**

* Considero que el sistema, en este caso, debería configurarse para reintentos automáticos limitados, por medio del uso de herramientas como Spring Retry o Resilience4j
* Establecer un numero máximo de intentos por transacción, 3 por ejemplo.
* También se podría agregar un estado adicional, como PENDING, para marcar esas transacciones fallidas y luego reintentarlas por medio de un job programado que busque ese estado e intente enviarlas de nuevo.

**Análisis consulta SQL**

La consulta usa un CASE para traducir claves UUID en etiquetas como ‘Merchandise’, ‘Cash’ o ‘not running’

Luego se encuentra un count(\*) quantity, lo que sugiere que se desea contar por tipo de producto.

También puedo observar que se están uniendo tablas relacionadas con solicitudes (user\_request, credit\_request, sales\_evaluation, etc.) esto para tener un contexto lo más completo posible de cada evaluación de venta o crédito. La clave principal en estos JOIN parece ser sales\_id, user\_id.

Luego de los JOIN observo un filtro de fecha para limitar los resultados entre el 1 y el 14 de agosto de 2022, es decir que se trata de un reporte quincenal.

Finalmente se agrupan por tipo de producto para contar cuantas solicitudes se generan por cada uno durante el periodo dado.