Sistemas Transaccionales
Proyecto - Entrega 3
Cristian Rugeles 202411069 y Valeria Martínez 202410228

1. Implementación de la Aplicación Transaccional (RF1-RF11)
Para esta entrega, se ha asegurado la atomicidad e integridad de todas las operaciones de negocio mediante la correcta implementación de la gestión de transacciones de Spring.
En la Entrega 2, se implementaron mejoras significativas en la gestión de transacciones y se corrigieron varios problemas identificados en la Entrega 1. Todos los requerimientos funcionales ahora operan dentro de transacciones atómicas que garantizan consistencia de datos.
Se revisaron todas las clases en el paquete service y se confirmó que todos los métodos que realizan operaciones de escritura (INSERT, UPDATE) en la base de datos están anotados con @Transactional. Esto garantiza que cada requisito funcional se ejecute como una unidad de trabajo atómica, previniendo datos corruptos o estados inconsistentes en la base de datos.

2. Implementación Transaccional de RF8 (Solicitar Servicio)
El Requerimiento Funcional 8 (Solicitar Servicio) se implementó como una única transacción atómica en el método crearServicio de la clase ServicioService.
La anotación @Transactional asegura que si cualquiera de los pasos listados a continuación falla, la transacción completa se revierte (ROLLBACK), sin dejar registros inconsistentes en la base de datos.

- Verificar Medio de Pago: Se valida que el número de tarjeta de crédito proporcionado exista y pertenezca al usuario solicitante, usando el método usuarioRepository.verificarTarjeta(...). Si falla, se lanza una AppLogicException y la transacción se aborta.

- Buscar Conductor Disponible: Se busca un conductor que esté disponible, usando la consulta servicioRepository.buscarConductorDisponible(...). La consulta SQL utiliza FOR UPDATE para aplicar un bloqueo pesimista y prevenir que dos solicitudes tomen al mismo conductor simultáneamente.

- Actualizar Estado del Conductor: Inmediatamente después de encontrar un conductor, se actualiza su estado a "OCUPADO" en la tabla USUARIO usando usuarioRepository.actualizarEstado(...). Esto lo saca de la lista de conductores disponibles para futuras solicitudes.

- Registrar el Viaje: Finalmente, se crea el nuevo registro en la tabla SERVICIO y sus tablas asociadas (DESTINOS_SERVICIO, TRANSPORTE_PASAJEROS, etc.).

Escenarios de Prueba de RF8
- Prueba de Éxito (COMMIT)

- Descripción: Se ejecuta una solicitud de servicio (RF8) con datos válidos de un usuario (2001) y su tarjeta de crédito correcta, según los datos de poblacion.sql.

- Resultado Esperado: La transacción se completa exitosamente (COMMIT). Se crea el servicio y el estado del conductor asignado se actualiza.

- Prueba de Falla (ROLLBACK)
- Descripción: Se ejecuta una solicitud de servicio (RF8) con una tarjeta de crédito inválida (1234) que no pertenece al usuario.
- Resultado Esperado: La transacción falla en el Paso 1 (Verificación de Tarjeta). La AppLogicException provoca un ROLLBACK.

3. Implementación de RFC1 con Niveles de Aislamiento

Se implementaron versiones especiales del RFC1 (Consultar Histórico) en ReporteService.java diseñadas para probar la concurrencia.
Ambos métodos ejecutan la consulta del historial, esperan 30 segundos (Thread.sleep(30000)) y vuelven a ejecutar la misma consulta, todo dentro de una única transacción.

- rfc1ConcurrenciaSerializable: Anotado con @Transactional(isolation = Isolation.SERIALIZABLE).

- rfc1ConcurrenciaReadCommitted: Anotado con @Transactional(isolation = Isolation.READ_COMMITTED).

4. Escenario de Concurrencia (Nivel SERIALIZABLE)

Pestaña 1 (Lector): Se accede a /rfc/historial y se solicita el historial del conductor 1001 con nivel SERIALIZABLE.

Pestaña 2 (Escritor): Mientras la Pestaña 1 está en la espera de 30 segundos, se accede a /rf/servicio y se solicita un nuevo servicio (usando usuario 2002) que es asignado al conductor 1001.

Descripción de lo Sucedido: La Pestaña 2 (RF8), que intenta ejecutar un INSERT en SERVICIO y un UPDATE en USUARIO (para el conductor 1001), fue bloqueada. La página se quedó cargando y no mostró "Éxito" hasta que la transacción de la Pestaña 1 (RFC1) completó sus 30 segundos y liberó el bloqueo sobre los datos leídos.

Resultado Presentado por RFC1: El nuevo servicio creado en la Pestaña 2 NO apareció en ninguna de las dos listas (ni "Antes" ni "Después").

Resultados Antes del Timer

ID Servicio	Tipo	Fecha Inicio	Fecha Fin	Costo	Conductor	Placa
202	TRANSPORTE_PASAJEROS	01/11/2025 19:52	01/11/2025 19:57	$20.000	Conductor 2	AUT002
102	TRANSPORTE_PASAJEROS	01/11/2025 01:52	01/11/2025 02:52	$17.000	Conductor 2	AUT002
2	TRANSPORTE_PASAJEROS	31/10/2025 15:52	31/10/2025 16:52	$17.000	Conductor 2	AUT002

Resultados Después del Timer

ID Servicio	Tipo	Fecha Inicio	Fecha Fin	Costo	Conductor	Placa
202	TRANSPORTE_PASAJEROS	01/11/2025 19:52	01/11/2025 19:57	$20.000	Conductor 2	AUT002
102	TRANSPORTE_PASAJEROS	01/11/2025 01:52	01/11/2025 02:52	$17.000	Conductor 2	AUT002
2	TRANSPORTE_PASAJEROS	31/10/2025 15:52	31/10/2025 16:52	$17.000	Conductor 2	AUT002

Esto demuestra el comportamiento de aislamiento SERIALIZABLE. La transacción de lectura (RFC1) operó sobre una snapshot de la base de datos tomada al inicio. Aisló completamente la transacción de los cambios realizados por RF8, previniendo el fenómeno de "lectura fantasma".

5. Escenario de Concurrencia (Nivel READ COMMITTED)
Pestaña 1 (Lector): Se accede a /rfc/historial y se solicita el historial del conductor 1002 con nivel READ COMMITTED.

Pestaña 2 (Escritor): Mientras la Pestaña 1 está en la espera de 30 segundos, se accede a /rf/servicio y se solicita un nuevo servicio (usando usuario 2003) que es asignado al conductor 1002.

Descripción de lo sucedido: La Pestaña 2 (RF8) NO fue bloqueada. La solicitud de servicio se procesó de inmediato y la transacción hizo COMMIT exitosamente, todo mientras la Pestaña 1 seguía en su espera de 30 segundos.

Resultado Presentado por RFC1: El nuevo servicio creado en la Pestaña 2 SÍ apareció en la lista "Resultados Después del Timer", pero no en la lista "Resultados Antes del Timer".

📊 Resultados Antes del Timer

ID Servicio	Tipo	Fecha Inicio	Fecha Fin	Costo	Conductor	Placa
1783	TRANSPORTE_PASAJEROS	01/11/2025 19:55	En curso	$20.000	Conductor 1	AUT001
201	TRANSPORTE_PASAJEROS	01/11/2025 19:52	01/11/2025 19:57	$20.000	Conductor 1	AUT001
101	TRANSPORTE_PASAJEROS	01/11/2025 00:52	01/11/2025 01:52	$16.000	Conductor 1	AUT001
1	TRANSPORTE_PASAJEROS	31/10/2025 14:52	31/10/2025 15:52	$16.000	Conductor 1	AUT001

📊 Resultados Después del Timer

ID Servicio	Tipo	Fecha Inicio	Fecha Fin	Costo	Conductor	Placa
1784	TRANSPORTE_PASAJEROS	01/11/2025 20:06	En curso	$20.000	Conductor 1	AUT001
1783	TRANSPORTE_PASAJEROS	01/11/2025 19:55	En curso	$20.000	Conductor 1	AUT001
201	TRANSPORTE_PASAJEROS	01/11/2025 19:52	01/11/2025 19:57	$20.000	Conductor 1	AUT001
101	TRANSPORTE_PASAJEROS	01/11/2025 00:52	01/11/2025 01:52	$16.000	Conductor 1	AUT001
1	TRANSPORTE_PASAJEROS	31/10/2025 14:52	31/10/2025 15:52	$16.000	Conductor 1	AUT001

Esto demuestra el fenómeno de Lectura No Repetible. El nivel READ COMMITTED solo garantiza que no se lean datos "sucios" (no confirmados). Sin embargo, permitió que la segunda consulta (después del timer) viera los datos que la Pestaña 2 acababa de confirmar (COMMIT), rompiendo la consistencia de la lectura dentro de la misma transacción.