Documentación

(*Santiago Orduz 202221218, Juan Francisco Rodríguez 202214603 & Felipe Lancheros 202211004*)

Actualización de requerimientos

Se realizaron actualizaciones significativas en la implementación de ciertos requerimientos funcionales de la API del sistema BancAndes. Estos cambios implicaron pasar de utilizar métodos CRUD básicos proporcionados por Spring Data JPA a construcciones de consultas SQL manuales para un control más detallado, una gestión transaccional mejorada y cumplir con la solicitud de llevar el manejo solo de forma manual.

1. Requerimiento de modificación 2 – Crear oficina (Punto Físico).

Se sustituyó el uso de métodos de Spring Data JPA para la creación de oficinas con punto físico con consultas SQL personalizadas en el repositorio OficinaRepository. Ahora se hace una implementación de consultas SQL manuales para la inserción de oficinas en la base de datos. Esto ya que en principio si se creó la consulta para una oficina normal, sin embargo, cuando esta tenía un punto de atención físico se tenía que ir al controller a revisar las validaciones correspondientes.

2. Requerimiento de modificación 3 – Crear y borrar puntos de atención.

El proceso de creación y eliminación de puntos de atención era gestionado por métodos automáticos, ahora se controla a través de consultas SQL manuales en PuntoDeAtencionRepository. Se desarrollaron consultas SQL para insertar y eliminar puntos de atención con verificaciones de existencia previas y validaciones de no existencia de operaciones vinculadas para permitir la eliminación.

3. **Requerimiento de modificación 6 – Registrar operación sobre cuenta (Transaccional).**

Las operaciones de consignación, retiro y transferencia de dinero de una cuenta anteriormente se manejaban a través de métodos del controlador basados en Spring Data JPA, ahora se gestionan mediante consultas SQL manuales en el repositorio de cuenta. Se implementaron métodos personalizados para insertar registros en el log de operaciones sobre cuentas y actualizar saldos de cuentas simultáneamente. Las validaciones que antes se hacían en todo el controller fueron movidas al servicio, de esta manera se harán todas las operaciones sin importar sus restricciones, cuando llega al servicio transaccional se pueden manejar las excepciones de forma que, si se cumplen las validaciones se hará el commit, de lo contrario habrá un rollback de la operación.

4. Requerimiento de modificación 8 – Registro de operación sobre préstamo.

Se añadieron consultas manuales para el registro de operaciones sobre préstamos, ahora se hace un manejo transaccional para asegurar la consistencia en el estado de los préstamos tras cada operación.

- **RF6 - Registrar Operación Sobre Cuenta**

  Método: insertOperacion

  Descripción: Inserta una nueva operación en la base de datos utilizando una consulta SQL nativa. Este método está marcado con @Modifying y @Transactional, lo que indica que modifica la base de datos y gestiona la transacción automáticamente.

  Parámetros:

  tipo: Tipo de operación (consignación, retiro, etc.).

  monto: Monto de la operación.

  fechaHora: Fecha y hora de la operación.

  puntoDeAtencion: Identificador del punto de atención donde se realiza la operación.

  producto: Identificador del producto asociado a la operación (cuenta bancaria).

  Consulta SQL: Inserta directamente en la tabla de operaciones, generando un nuevo ID automáticamente para cada operación.

- **RFC4 & RFC5 - Consulta de Operaciones Realizadas Sobre Una Cuenta**

  Método: findByCuentaAndFecha

  Descripción: Recupera todas las operaciones realizadas sobre una cuenta específica dentro de un rango de fechas. Este método puede usarse para implementar RFC4 (SERIALIZABLE) y RFC5 (READ COMMITTED) dependiendo de los ajustes de transaccionalidad en el servicio que lo invoque.

  Parámetros:

  numeroCuenta: Número de la cuenta.

  fechaInicio: Fecha de inicio del periodo de consulta.

  fechaFin: Fecha de fin del periodo de consulta.

  Consulta SQL: Utiliza una consulta nativa que filtra operaciones por número de cuenta y rango de fechas, solo incluyendo tipos de operación específicos como consignaciones, retiros y transferencias.

Estructura de Capas

1. Capa de Modelo

   La capa de modelo es donde se define la estructura de los datos que maneja la aplicación. Este nivel es crucial ya que establece las entidades base que se utilizan en todo el sistema, tales como Cuenta, GerenteOficina, Oficina, Operacion, Prestamo, Producto, PuntoDeAtencion, PuntoFisico, Ubicacion y Usuario. Cada una de estas clases está diseñada para mapear a una tabla específica en la base de datos utilizando anotaciones de JPA para facilitar la integración y manipulación de datos.

2. Capa de Repositorio

   La capa de repositorio contiene interfaces que extienden JpaRepository o similares, proporcionando métodos CRUD (Crear, Leer, Actualizar, Eliminar) esenciales junto con operaciones de base de datos personalizadas. Estos repositorios actúan como un puente entre la capa de datos y la capa de servicios, permitiendo operaciones más complejas que pueden incluir consultas personalizadas o transacciones más elaboradas.

3. Capa de Servicio

   La capa de servicios maneja la lógica de negocio y es responsable de definir las transacciones y los niveles de aislamiento requeridos para mantener la integridad y la consistencia de los datos. Por ejemplo, en CuentaService se definen métodos para gestionar las cuentas bancarias, incluyendo operaciones como transferencias, depósitos y retiros, cada uno con su propio control transaccional y lógica específica.

4. Capa de Controlador

   Los controladores en la aplicación son responsables de manejar las interacciones del usuario a través de la web. Definen los endpoints de la API, aceptan solicitudes HTTP, y llaman a los servicios adecuados basándose en la acción del usuario. Además, gestionan las respuestas que se deben enviar al cliente, manejando también los posibles errores que pueden surgir durante el procesamiento de las solicitudes.

Integración y Funcionamiento General

Cada capa está diseñada para funcionar de manera interdependiente, asegurando que los datos fluyan correctamente desde la interfaz de usuario hasta la persistencia de datos. Los

controladores reciben y manejan las solicitudes, pasando el control a los servicios que aplican la lógica de negocio necesaria. Los servicios utilizan los repositorios para interactuar con la base de datos, y los modelos sirven como estructuras de datos que representan las tablas de la base de datos.

Esta estructura de múltiples capas permite separar claramente las responsabilidades, facilita el mantenimiento y la escalabilidad del sistema, y ayuda a proteger la integridad de los datos mediante la gestión adecuada de transacciones y niveles de aislamiento en las operaciones críticas.

Capa de Controller: Integración y Tests de Transaccionalidad

1. consultarOperacionesSerializable(int numeroCuenta)

   Nivel de Aislamiento: SERIALIZABLE

   Descripción: Recupera las operaciones de una cuenta durante el último mes. Este nivel de aislamiento asegura que la consulta sea completamente aislada de otras transacciones, evitando lecturas sucias y no repetibles, así como problemas de phantoms.

   Uso práctico: Ideal para garantizar vistas consistentes en situaciones donde la precisión de los datos en cada consulta es crítica, como en informes financieros o auditorías.

2. consultarOperacionesReadCommitted(int numeroCuenta)

   Nivel de Aislamiento: READ COMMITTED

   Descripción: Similar al método anterior pero con un nivel de aislamiento que permite ver los cambios confirmados por otras transacciones, lo que es útil para operaciones que requieren menos rigidez y mayor rendimiento.

   Uso práctico: Adecuado para aplicaciones en tiempo real donde la información puede ser ligeramente menos precisa, pero se requiere mayor rapidez.

3. consignarDinero(Long cuentaId, Double monto)

   Nivel de Aislamiento: Default (generalmente READ COMMITTED en muchas bases de datos)

   Descripción: Añade dinero a una cuenta, validando primero que la cuenta exista y esté activa, y que el monto sea positivo.

Uso práctico: Se utiliza para aumentar el saldo de una cuenta, con rollback automático en caso de errores, garantizando la consistencia de los datos.

4. retirarDinero(Long cuentaId, Long puntoDeAtencionId, Double monto)

   Nivel de Aislamiento: Default

   Descripción: Permite a un usuario retirar dinero de su cuenta, asegurándose de que la cuenta tenga fondos suficientes y que el punto de atención permita el retiro.

   Uso práctico: Es crucial para gestionar los retiros de manera segura, evitando que los usuarios retiren más de lo que tienen o que realicen operaciones en puntos no autorizados.

5. transferirDinero(Long cuentaOrigenId, Long cuentaDestinoId, Long puntoDeAtencionId, Double monto)

   Nivel de Aislamiento: Default

   Descripción: Transfiere fondos de una cuenta a otra, validando que ambas cuentas estén activas, que el punto de atención permita la transferencia y que haya fondos suficientes en la cuenta de origen.

   Uso práctico: Facilita las transferencias entre cuentas con un control riguroso para evitar sobregiros y errores de transferencia, asegurando transacciones limpias y seguras.

Consideraciones Generales de Transaccionalidad

Manejo de Excepciones y Rollback: Todos los métodos que realizan cambios significativos en la base de datos están diseñados para manejar excepciones adecuadamente. Utilizamos TransactionAspectSupport.currentTransactionStatus().setRollbackOnly() para marcar una transacción para rollback si se detecta una condición de error, garantizando que no se persistan cambios parciales o incorrectos.

Validaciones: Antes de realizar cualquier operación significativa, se llevan a cabo validaciones para asegurar que las operaciones se realizan en el contexto adecuado (por ejemplo, verificar que una cuenta esté activa o que los montos sean positivos), lo que añade una capa adicional de seguridad y robustez a las operaciones.

Capa de Controller: Integración y Tests de Transaccionalidad

1. getAllCuentas()

   Tipo: GET

Ruta: “/cuentas”

Descripción: Retorna todas las cuentas bancarias disponibles.

Transaccionalidad: Realiza una operación de lectura no transaccional, mostrando la lista de cuentas sin aplicar cambios.

2. puntosDeAtencion(Model model)

   Tipo: GET

   Ruta: “/cuentas/cuenta”

   Descripción: Carga datos de las cuentas en el modelo para mostrar en la vista cuentas.

   Transaccionalidad: No realiza cambios en la base de datos; su función principal es de visualización.

3. consignacionForm(Model model)

   Tipo: GET

   Ruta: “/cuentas/consignacion”

   Descripción: Prepara el formulario para realizar una consignación, añadiendo objetos necesarios al modelo.

   Transaccionalidad: No hay transacciones involucradas ya que sólo prepara la vista para la entrada de datos.

4. hacerConsignacion(@ModelAttribute Operacion operacion,

   @ModelAttribute Producto producto, @ModelAttribute Object[] datos)

   Tipo: GET (debería ser POST para procesar datos)

   Ruta: “/cuentas/consignacion/save”

   Descripción: Procesa la consignación especificada, idealmente insertando la operación en la base de datos.

   Transaccionalidad: Aunque no se detalla el manejo de transacciones, este método debería utilizar un nivel de aislamiento adecuado para garantizar que la operación de consignación se registre de manera segura y coherente.

5. transferenciaForm(Model model)

   Tipo: GET

Ruta: “/cuentas/transferencia”

Descripción: Prepara el formulario para realizar una transferencia entre cuentas.

Transaccionalidad: Similar al formulario de consignación, prepara la vista sin realizar operaciones transaccionales.

6. hacerTransferencia(@ModelAttribute Operacion operacion, @ModelAttribute Producto producto, @ModelAttribute Object[] datos)

   Tipo: GET (debería ser POST para procesar datos)

   Ruta: “/cuentas/transferencia/save”

   Descripción: Ejecuta una transferencia de fondos basada en los datos suministrados.

   Transaccionalidad: Este método debe manejar transacciones para asegurar que la transferencia se complete sin interferencias y con integridad de datos.

7. cerrarPrestamo(Long id)

   Tipo: GET

   Ruta: “/cuentas/cuenta/{id}/cerrar”

   Descripción: Cambia el estado de una cuenta a cerrada.

   Transaccionalidad: Requiere manejo transaccional para asegurar que el cambio de estado sea persistente y coherente.

8. desactivarPrestamo(Long id)

   Tipo: GET

   Ruta: “/cuentas/cuenta/{id}/desactivar”

   Descripción: Cambia el estado de una cuenta a desactivada.

   Transaccionalidad: Similar a cerrar préstamo, debe garantizar la atomicidad del cambio de estado.

9. getCuentaById(Long id)

   Tipo: GET

   Ruta: “/cuentas/{id}”

   Descripción: Obtiene detalles de una cuenta específica.

Transaccionalidad: Asegura que la cuenta obtenida es actual y no afectada por transacciones simultáneas.

10. cuentaForm(Model model)

    Tipo: GET

    Ruta: “/cuentas/new”

    Descripción: Prepara el formulario para crear una nueva cuenta.

    Transaccionalidad: No aplica transacciones ya que solo prepara la vista.

11. cuentaGuardar(@ModelAttribute Cuenta cuenta, @ModelAttribute Usuario usuario)

    Tipo: POST

    Ruta: “/cuentas/new/save”

    Descripción: Guarda la nueva cuenta y usuario en la base de datos.

    Transaccionalidad: Debe usar transacciones para asegurar la creación coherente y segura de los registros.

12. updateCuenta(Long id, @RequestBody Cuenta cuentaDetails)

    Tipo: PUT

    Ruta: “/cuentas/{id}”

    Descripción: Actualiza los detalles de una cuenta existente.

    Transaccionalidad: Maneja transacciones para asegurar que los cambios sean coherentes y aislados.

13. deleteCuenta(Long id)

    Tipo: DELETE

    Ruta: “/cuentas/{id}”

    Descripción: Elimina una cuenta existente.

    Transaccionalidad: Utiliza transacciones para garantizar que la eliminación sea definitiva y segura.

Escenarios de Prueba de Concurrencia

Escenario #1:

Escenario: Un usuario ejecuta primero el RFC4 - CONSULTA DE OPERACIONES REALIZADAS SOBRE UNA CUENTA – SERIALIZABLE, que busca las operaciones realizadas en una cuenta en el último mes. Mientras esta consulta aún está en ejecución, el mismo usuario intenta realizar una consignación usando RF6 - REGISTRAR OPERACIÓN SOBRE CUENTA (versión transaccional) de manera concurrente.

Pasos para la Ejecución Concurrente de RFC4 y RF6

Inicio de RFC4: El usuario inicia la consulta de operaciones realizadas sobre una cuenta (RFC4), especificando el número de cuenta y un rango de fechas que incluye el último mes. Esta consulta se ejecuta con un nivel de aislamiento SERIALIZABLE.

Ejecución Concurrente de RF6: Antes de que finalicen los 30 segundos de la consulta RFC4, el usuario inicia una consignación sobre la misma cuenta a través de RF6. Este intento de transacción también implica la modificación del saldo de la cuenta.

Descripción de lo Sucedido

Bloqueo de la Consignación: Dado que la consulta RFC4 está ejecutándose con un nivel de aislamiento SERIALIZABLE, esta mantiene un bloqueo sobre los datos que consulta, incluyendo el saldo de la cuenta. Cualquier intento de modificar esos datos, como la consignación en RF6, debe esperar a que finalice RFC4.

Espera de RF6: El componente que implementa RF6 (consignación) debe esperar a que la consulta RFC4 termine y libere los bloqueos sobre los datos de la cuenta. Esta espera es necesaria para mantener la integridad de los datos y evitar conflictos de transacciones.

Resultado Presentado por RFC4

Resultado de RFC4: La consulta RFC4 mostrará todas las operaciones realizadas en el último mes, sin incluir la consignación realizada concurrentemente en RF6 si esta se inició después de que la consulta comenzara. Esto se debe al nivel de aislamiento SERIALIZABLE, que garantiza que la transacción vea solo datos que no han sido modificados durante su ejecución.

Aparición de la Consignación en RFC4: No, la consignación que se intentó realizar durante la ejecución de RFC4 no aparecerá en los resultados de esta consulta. Solo será visible en consultas futuras una vez que la consignación haya sido completada y confirmada.

Escenario #2:

Escenario: Un usuario inicia la consulta de operaciones realizadas sobre una cuenta (RFC5), que utiliza el nivel de aislamiento READ COMMITTED. Durante esta consulta, el usuario intenta realizar una consignación en la misma cuenta utilizando RF6 - REGISTRAR OPERACIÓN SOBRE CUENTA.

Pasos para la Ejecución Concurrente de RFC5 y RF6

Inicio de RFC5: El usuario comienza la consulta de las operaciones realizadas en una cuenta durante el último mes, bajo el nivel de aislamiento READ COMMITTED.

Ejecución Concurrente de RF6: Antes de que finalicen los 30 segundos desde el inicio de RFC5, el usuario intenta realizar una consignación en la misma cuenta. Esta acción intenta modificar el saldo de la cuenta.

Descripción de lo Sucedido

Ejecución de RF6 Durante RFC5: Bajo READ COMMITTED, la consulta en RFC5 solo bloquea las filas leídas para evitar lecturas sucias, pero permite que otras transacciones modifiquen estas filas siempre que no intenten leerse nuevamente durante la misma transacción. Por lo tanto, RF6 puede proceder con la consignación sin necesidad de esperar a que termine RFC5.

Transacciones: Dado que READ COMMITTED no previene por completo las interacciones entre las transacciones concurrentes, RF6 podría modificar con éxito los datos (el saldo de la cuenta) mientras RFC5 sigue en ejecución.

Resultado Presentado por RFC5

Resultado de RFC5: La consulta mostrará todas las operaciones realizadas en el último mes hasta el momento de su inicio. Como RF6 se ejecuta tras el inicio de RFC5, y RFC5 usa READ COMMITTED, no se garantiza que la consignación hecha por RF6 sea visible en los resultados de RFC5 si la consulta no vuelve a leer los datos modificados tras comenzar RF6.

Aparición de la Consignación en RFC5: La consignación realizada no aparece en los resultados de RFC5, ya que los datos modificados por RF6 no se leen nuevamente por la consulta de RFC5 tras la modificación.